

Flink 1.15.2



Apache Flink

1. Flink 框架介绍

1.1. 发展历史

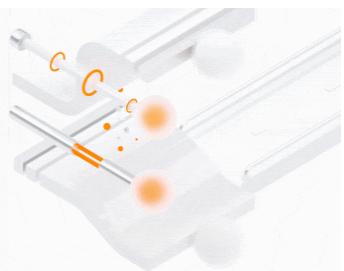
- Flink 诞生于欧洲的一个大数据研究项目 StratoSphere。它是由 3 所地处柏林的大学和欧洲其他一些大学在 2010~2014 年共同进行的研究项目，由柏林理工大学的教授沃克尔·马尔科 (Volker Markl) 领衔开发。
- 早期，Flink 是做 Batch 计算的，但是在 2014 年，StratoSphere 里面的核心成员孵化出 Flink，同时 Flink 计算的主流方向被定位为 Streaming
- 2014 年 8 月，Flink 第一个版本 0.6 正式发布（至于 0.5 之前的版本，那就是在 Stratosphere 名下的了）。与此同时 Flink 的几位核心开发者创办了 Data Artisans 公司，主要做 Flink 的商业应用，帮助企业部署大规模数据处理解决方案。
- 2014 年 12 月，Flink 项目完成了孵化，一跃成为 Apache 软件基金会的顶级项目。
- 2015 年 4 月，Flink 发布了里程碑式的重要版本 0.9.0，很多国内外大公司也正是从这时开始关注、并参与到 Flink 社区建设的。
- 2019 年 1 月，长期对 Flink 投入研发的阿里巴巴，以 9000 万欧元的价格收购了 Data Artisans 公司；之后又将自己的内部版本 Blink 开源，继而与 8 月份发布的 Flink 1.9.0 版本进行了合并。自此之后，Flink 被越来越多的人所熟知，成为当前最火的新一代大数据处理框架。
- 阿里云 Flink 产品
 - <https://www.aliyun.com/product/bigdata/sc>

实时计算Flink版 播放视频

实时计算Flink版 (Alibaba Cloud Realtime Compute for Apache Flink, Powered by Ververica) 是阿里云基于 Apache Flink 构建的企业级、高性能实时大数据处理系统，由 Apache Flink 创始团队官方出品，拥有全球统一商业化品牌，完全兼容开源 Flink API，提供丰富的企业级增值功能。

按量付费新客限时7折，结合Autopilot调优使用效果更佳！点此立即申请

[立即开通](#) [管理控制台](#) [文档中心](#) [免费上手体验](#) [产品快速入门](#) [Flink中文社区](#)

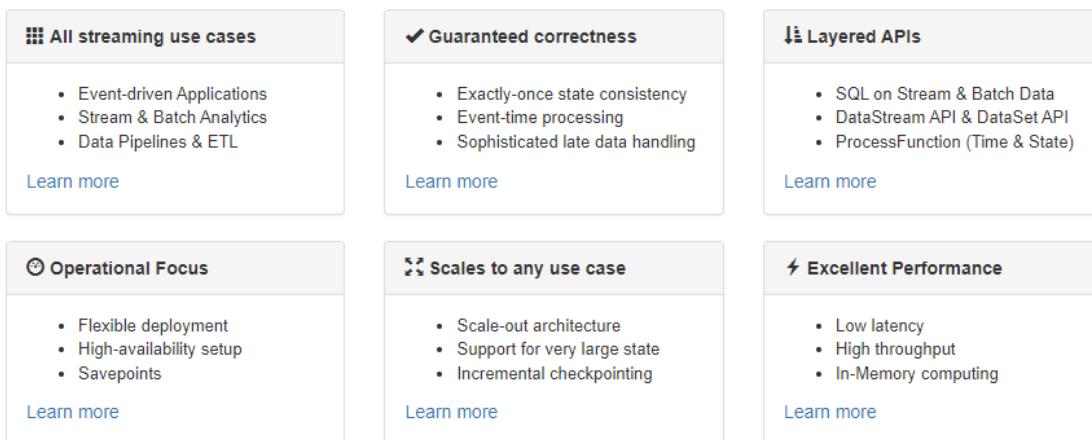
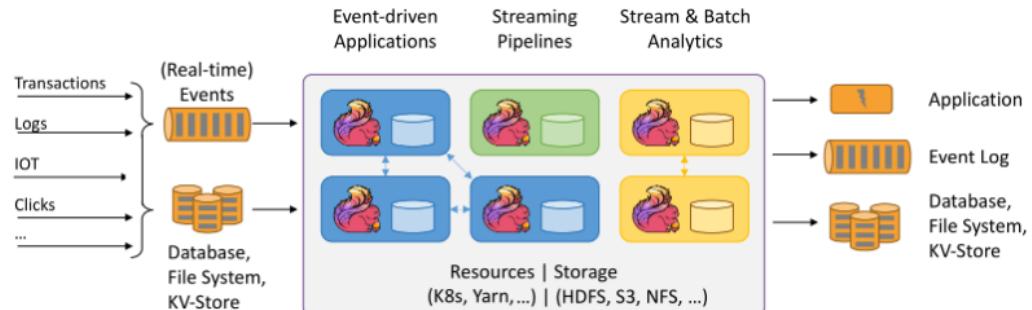


1.2. 官网地址

- Flink 的官网主页地址：<https://flink.apache.org/>

- Flink 的中文主页地址: <https://flink.apache.org/zh/>
- Flink 的中文社区地址: <https://flink-learning.org.cn/activity>
- 阿里云Flink技术地址: <https://help.aliyun.com/product/45029.html>

- **Apache Flink® — Stateful Computations over Data Streams**



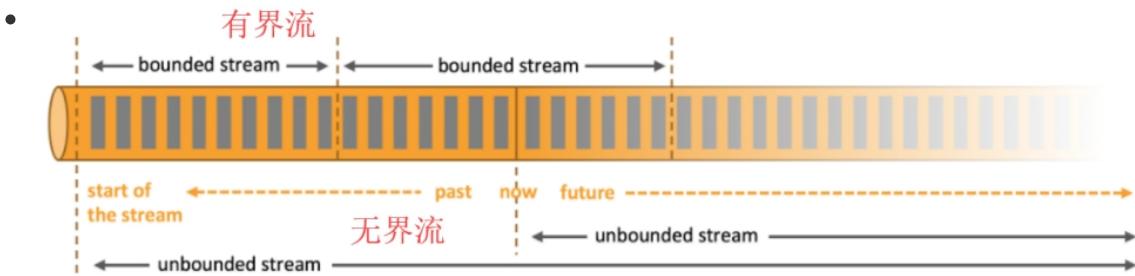
- 框架优势:

- 所有流式场景
- 正确性保证
- 分层API
- 聚集运维
- 大规模运算
- 性能卓越

1.3. 无界和有界数据

- 任何类型的数据都可以形成一种事件流。信用卡交易、传感器测量、机器日志、网站或移动应用程序上的用户交互记录，所有这些数据都形成一种流。
- 数据可以被作为 无界 或者 有界 流来处理。
 - **无界流**
 - 有定义流的开始，但没有定义流的结束。它们会无休止地产生数据。无界流的数据必须持续处理，即数据被摄取后需要立刻处理。
 - 我们不能等到所有数据都到达再处理，因为输入是无限的，在任何时候输入都不会完成。
 - 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，以便能够推断结果的完整性。
 - **有界流**
 - 有定义流的开始，也有定义流的结束。有界流可以在摄取所有数据后再进行计算。

- 有界流所有数据可以被排序，所以并不需要有序摄取。
- 有界流处理通常被称为批处理



- Apache Flink 擅长处理无界和有界数据集

1.4. 学习Flink前置技能

- Java 新增特性 Lambda表达式
- 安装JDK 11编程语言
- 安装Scala 2.12.x编程语言
- 熟练掌握SQL语句和各种常用函数
- Kafka 0.10.x 以上
- Hadoop 3.x.x 以上

2. Flink 编程模型

2.1. 分层api

- | | | |
|------------------------------------|---------------------------------------|----------------------------------|
| High-level Analytics API | SQL / Table API (dynamic tables) | - Conciseness + Expressiveness - |
| Stream- & Batch Data Processing | DataStream API (streams, windows) | + Expressiveness + |
| Stateful Event-Driven Applications | ProcessFunction (events, state, time) | |
- Stateful Stream Processing
 - ProcessFunction是Flink最底层的接口。
 - ProcessFunction可以处理一或者两条输入数据流中的单个事件或者归入一个特定窗口内的多个事件。
 - 它提供了对时间和状态的细粒度控制。
 - 虽然灵活性高，但开发比较复杂，需要具备一定的编码能力。
- DataStream DataSet API
 - DataStream API为许多通用的流处理操作提供了原语，其实就是在ProcessFunction的基础上多了一些算子。
 - DataStream API 支持Java 和 Scala 语言，预先定义了例如`map()`、`reduce()`、`aggregate()`等函数。
 - DataSet API 是批处理API，处理有限的数据集。
 - DataStream API是流处理API，处理无限的数据集。

- SQL&Table API:
 - SQL 构建在Table 之上，都需要构建Table 环境。
 - Table 可以与DataStream或者DataSet进行相互转换。
 - Streaming SQL不同于存储的SQL，最终会转化为流式执行计划
- 示例代码，分别为ProcessFunction、DataStream API、SQL&Table API

```

/*
 * 将相邻的 keyed START 和 END 事件相匹配并计算两者的时间间隔
 * 输入数据为 Tuple2<String, String> 类型，第一个字段为 key 值,
 * 第二个字段标记 START 和 END 事件。
 */
public static class StartEndDuration
    extends KeyedProcessFunction<String, Tuple2<String, String>, Tuple2<String, Long>> {

    private ValueState<Long> startTime;

    @Override
    public void open(Configuration conf) {
        // obtain state handle
        startTime = getRuntimeContext()
            .getSTATE(new ValueStateDescriptor<Long>("startTime", Long.class));
    }

    /** Called for each processed event. */
    @Override
    public void processElement(
        Tuple2<String, String> in,
        Context ctx,
        Collector<Tuple2<String, Long>> out) throws Exception {

        switch (in.f1) {
            case "START":
                // set the start time if we receive a start event.
                startTime.update(ctx.timestamp());
                // register a timer in four hours from the start event.
                ctx.timerService()
                    .registerEventTimeTimer(ctx.timestamp() + 4 * 60 * 60 * 1000);
                break;
            case "END":
                // emit the duration between start and end event
                Long sTime = startTime.value();
                if (sTime != null) {
                    out.collect(Tuple2.of(in.f0, ctx.timestamp() - sTime));
                    // clear the state
                    startTime.clear();
                }
            default:
                // do nothing
        }
    }

    /** Called when a timer fires. */
    @Override
    public void onTimer(
        long timestamp,
        OnTimerContext ctx,
        Collector<Tuple2<String, Long>> out) {

        // Timeout interval exceeded. Cleaning up the state.
        startTime.clear();
    }
}

```

```

// 网站点击 Click 的数据流
DataStream<Click> clicks = ...;

DataStream<Tuple2<String, Long>> result = clicks
    // 将网站点击映射为 (userId, 1) 以便计数
    .map(
        // 实现 MapFunction 接口定义函数
        new MapFunction<Click, Tuple2<String, Long>>() {
            @Override
            public Tuple2<String, Long> map(Click click) {
                return Tuple2.of(click.userId, 1L);
            }
        })
    // 以 userId (field 0) 作为 key
    .keyBy(0)
    // 定义 30 分钟超时的会话窗口
    .window(EventTimeSessionWindows.withGap(Time.minutes(30L)))
    // 对每个会话窗口的点击进行计数，使用 Lambda 表达式定义 reduce 函数
    .reduce((a, b) -> Tuple2.of(a.f0, a.f1 + b.f1));

```



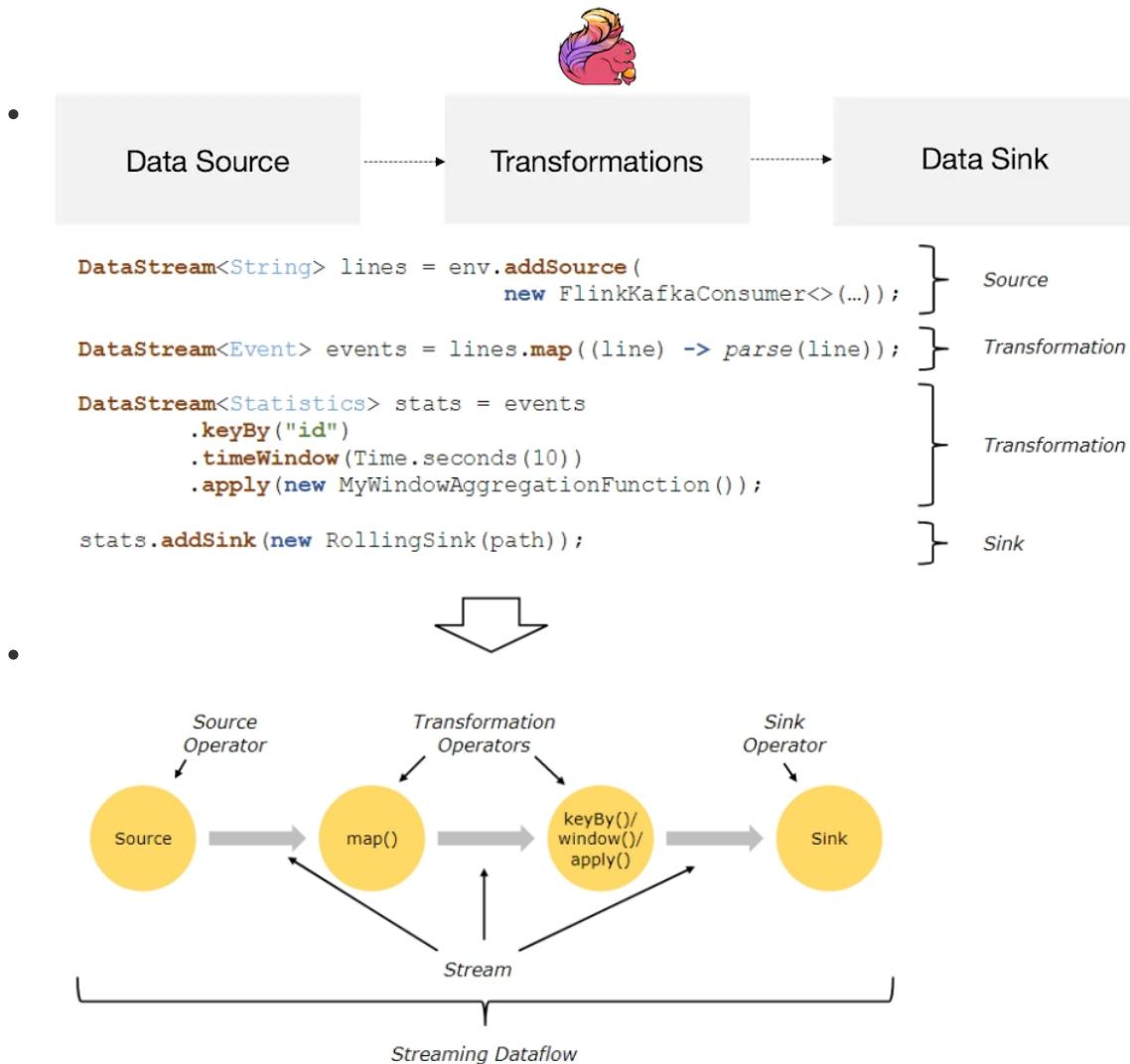
```

SELECT userId, COUNT(*)
FROM clicks
GROUP BY SESSION(clicktime, INTERVAL '30' MINUTE), userId

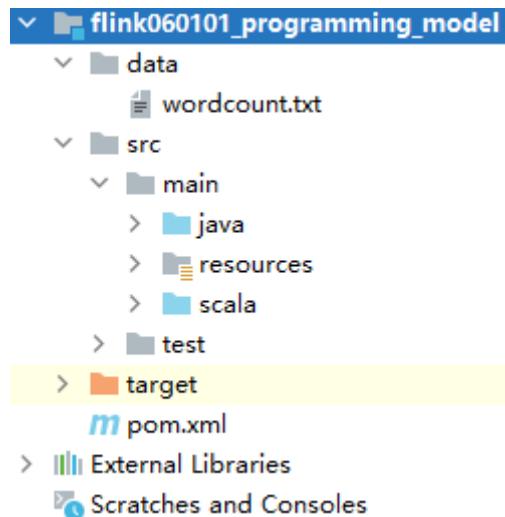
```

2.2. 编程模型

- 每个flink程序由source operator + transformation operator + sink operator组成



2.3. 项目搭建



2.3.1. maven依赖

- 创建一个Maven版本的Scala项目，Flink1.15.2版本需求【Java 11】【Scala 2.12.x】
- Flink基本依赖

- ```

<dependency>
 <groupId>org.apache.flink</groupId>
 <artifactId>flink-java</artifactId>
 <version>1.15.2</version>
</dependency>
<dependency>
 <groupId>org.apache.flink</groupId>
 <artifactId>flink-streaming-java</artifactId>
 <version>1.15.2</version>
</dependency>
<dependency>
 <groupId>org.apache.flink</groupId>
 <artifactId>flink-clients</artifactId>
 <version>1.15.2</version>
</dependency>

```

- Scala API:**

为了使用 Scala API，将 `flink-java` 的 artifact id 替换为 `flink-scala_2.12`，同时将 `flink-streaming-java` 替换为 `flink-streaming-scala_2.12`

- 项目所需要依赖

```

<properties>
 <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
 <maven.compiler.source>11</maven.compiler.source>
 <maven.compiler.target>11</maven.compiler.target>
 <flink.version>1.15.2</flink.version>
 <scala.version>2.12.11</scala.version>
 <log4j.version>2.12.1</log4j.version>
</properties>

<dependencies>
 <!-- Java开发环境 -->

```

```
<dependency>
 <groupId>org.apache.flink</groupId>
 <artifactId>flink-java</artifactId>
 <version>${flink.version}</version>
</dependency>
<dependency>
 <groupId>org.apache.flink</groupId>
 <artifactId>flink-streaming-java</artifactId>
 <version>${flink.version}</version>
</dependency>
<!-- Scala开发环境 -->
<dependency>
 <groupId>org.apache.flink</groupId>
 <artifactId>flink-scala_2.12</artifactId>
 <version>${flink.version}</version>
</dependency>
<dependency>
 <groupId>org.apache.flink</groupId>
 <artifactId>flink-streaming-scala_2.12</artifactId>
 <version>${flink.version}</version>
</dependency>
<!-- Flink客户端连接 -->
<dependency>
 <groupId>org.apache.flink</groupId>
 <artifactId>flink-clients</artifactId>
 <version>${flink.version}</version>
</dependency>

<!-- 本地运行, webUI服务的依赖 -->
<dependency>
 <groupId>org.apache.flink</groupId>
 <artifactId>flink-runtime-web</artifactId>
 <version>${flink.version}</version>
</dependency>

<!-- flink与Kafka整合的依赖 -->
<dependency>
 <groupId>org.apache.flink</groupId>
 <artifactId>flink-connector-kafka</artifactId>
 <version>${flink.version}</version>
</dependency>
<dependency>
 <groupId>org.apache.flink</groupId>
 <artifactId>flink-connector-base</artifactId>
 <version>${flink.version}</version>
</dependency>
<!-- flink与JDBC整合的依赖 -->
<dependency>
 <groupId>org.apache.flink</groupId>
 <artifactId>flink-connector-jdbc</artifactId>
 <version>${flink.version}</version>
</dependency>
<dependency>
 <groupId>mysql</groupId>
 <artifactId>mysql-connector-java</artifactId>
 <version>8.0.18</version>
</dependency>
<!-- 状态后端 rocksdb-->
```

```

<dependency>
 <groupId>org.apache.flink</groupId>
 <artifactId>flink-statebackend-rocksdb</artifactId>
 <version>${flink.version}</version>
</dependency>

<!-- 日志系统-->
<dependency>
 <groupId>org.apache.logging.log4j</groupId>
 <artifactId>log4j-slf4j-impl</artifactId>
 <version>${log4j.version}</version>
 <scope>runtime</scope>
</dependency>

<dependency>
 <groupId>org.apache.logging.log4j</groupId>
 <artifactId>log4j-api</artifactId>
 <version>${log4j.version}</version>
 <scope>runtime</scope>
</dependency>

<dependency>
 <groupId>org.apache.logging.log4j</groupId>
 <artifactId>log4j-core</artifactId>
 <version>${log4j.version}</version>
 <scope>runtime</scope>
</dependency>
</dependencies>

```

### 2.3.2. Log4j 配置

```

#####
#####
Licensed to the Apache Software Foundation (ASF) under one
or more contributor license agreements. See the NOTICE file
distributed with this work for additional information
regarding copyright ownership. The ASF licenses this file
to you under the Apache License, Version 2.0 (the
"License"); you may not use this file except in compliance
with the License. You may obtain a copy of the License at
#
http://www.apache.org/licenses/LICENSE-2.0
#
Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
#####
#####

rootLogger.level = WARN
rootLogger.appenderRef.console.ref = ConsoleAppender

appender.console.name = ConsoleAppender
appender.console.type = CONSOLE
appender.console.layout.type = PatternLayout

```

```
appender.console.layout.pattern = %d{HH:mm:ss,SSS} %-5p %-60c %x - %m%n
```

### 2.3.3. 目标数据

```
hello01 yjxxt01
hello02 yjxxt02
hello03 yjxxt03
hello04 yjxxt04
hello05 yjxxt05
hello06 yjxxt06
hello07 yjxxt07
hello08 yjxxt08
hello01 yjxxt01
hello02 yjxxt02
hello03 yjxxt03
hello04 yjxxt04
hello05 yjxxt05
hello06 yjxxt06
hello07 yjxxt07
hello08 yjxxt08
```

## 2.4. DataSet版WordCount

### 2.4.1. Java代码实现

```
import org.apache.flink.api.common.functions.FlatMapFunction;
import org.apache.flink.api.common.functions.MapFunction;
import org.apache.flink.api.java.DataSet;
import org.apache.flink.api.java.ExecutionEnvironment;
import org.apache.flink.api.java.operators.AggregateOperator;
import org.apache.flink.api.java.operators.FlatMapOperator;
import org.apache.flink.api.java.operators.MapOperator;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.util.Collector;

import java.util.Arrays;

/**
 * @Description :
 * @School:优极限学堂
 * @Official-website: http://www.yjxxt.com
 * @Teacher:李毅大帝
 * @Mail:863159469@qq.com
 */
public class Hello01WordCountByDataSetUseJava {
 public static void main(String[] args) throws Exception {
 //创建环境
 ExecutionEnvironment environment =
 ExecutionEnvironment.getExecutionEnvironment();
 //获取数据源
 DataSet<String> source =
 environment.readTextFile("data/wordcount.txt");
 //开始转换
```

```

 FlatMapOperator<String, String> flatMap = source.flatMap(new
FlatMapFunction<String, String>() {
 @Override
 public void flatMap(String line, collector<String> collector)
throws Exception {
 Arrays.stream(line.split(" ")).forEach(word ->
collector.collect(word));
 }
});
//开始计数
MapOperator<String, Tuple2<String, Integer>> map = flatMap.map(new
MapFunction<String, Tuple2<String, Integer>>() {
 @Override
 public Tuple2<String, Integer> map(String word) throws Exception
{
 return Tuple2.of(word, 1);
 }
});
//开始分类并统计
AggregateOperator<Tuple2<String, Integer>> sum =
map.groupBy(0).sum(1);

//打印出结果
sum.print();
}
}

```

## 2.4.2. scala代码实现

- ```

import org.apache.flink.api.scala._

/**
 * @Description :
 * @School:优极限学堂
 * @official-website: http://www.yjxxt.com
 * @Teacher:李毅大帝
 * @Mail:863159469@qq.com
 */
object Hello02WordCountByDataSetUseScala {
    def main(args: Array[String]): Unit = {
        //创建环境
        val environment = ExecutionEnvironment.getExecutionEnvironment;
        //读取数据源
        val source =
environment.readTextFile("src/main/resources/wordcount.txt");
        //开始转换
        val wordSet = source.flatMap(_.split(" "))
        val mapSet = wordSet.map((_, 1))
        //开始计数
        val groupBySet = mapSet.groupBy(0)
        //开始分类并统计
        val sumSet = groupBySet.sum(1)
        //打印出结果
        sumSet.print()
    }
}

```

2.5. DataStream版WordCount

2.5.1. 安装netcat.ext

- Windows
 - 下载软件 <https://eternallybored.org/misc/netcat/>
Here's [netcat 1.11](#) compiled for both 32 and 64-bit Windows (but note that 64-bit version hasn't been tested much - use at your own risk).
I'm providing it here because I never seem to be able to find a working netcat download when I need it.
 - Small update: [netcat 1.12](#) - adds -c command-line option to send CRLF line endings instead of just CR (eg. to talk to Exchange SMTP)
Warning: a bunch of antivirus think that netcat (nc.exe) is harmful for some reason, and may block or delete the file when you try to download it. I could get around this by recompiling the binary every now and then (without doing any other changes at all, which should give you an idea about the level of protection these products offer), but I really can't be bothered.



- 将下载后nc.exe和nc64.exe的软件存放到 C:\Windows\System32 目录下
 - 打开Doc窗口，执行命令 nc -l -p 19523(通信端口)
- Linux
 - [root@node01 ~]# yum install nc -y
 - [root@node01 ~]# nc -l -k -p 19523

2.5.2. Java代码实现

```
import org.apache.flink.api.common.functions.FlatMapFunction;
import org.apache.flink.api.common.functions.MapFunction;
import org.apache.flink.api.java.functions.KeySelector;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.streaming.api.datastream.DataStreamSource;
import org.apache.flink.streaming.api.datastream.KeyedStream;
import org.apache.flink.streaming.api.datastream.SingleOutputStreamOperator;
import
org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.util.Collector;

/**
 * @Description :
 * @School:优极限学堂
 * @Official-Website: http://www.yjxxt.com
 * @Teacher:李毅大帝
 * @Mail:863159469@qq.com
 */
public class Hello03WordCountByDataStreamUseJava {
    public static void main(String[] args) throws Exception {
        //获取程序运行的环境
        StreamExecutionEnvironment environment =
        StreamExecutionEnvironment.getExecutionEnvironment();
        //调用Source方法创建DataStream
        DataStreamSource<String> lineStream =
        environment.socketTextStream("localhost", 19523);

        //调用Transformation(s)
```

```

        singleOutputStreamOperator<String> wordStream =
lineStream.flatMap(new FlatMapFunction<String, String>() {
    @Override
    public void flatMap(String line, collector<String> collector)
throws Exception {
    String[] words = line.split(" ");
    for (String word : words) {
        collector.collect(word);
    }
}
});

SingleOutputStreamOperator<Tuple2<String, Integer>> mapStream =
wordStream.map(new MapFunction<String, Tuple2<String, Integer>>() {
    @Override
    public Tuple2<String, Integer> map(String w) throws Exception {
        return Tuple2.of(w, 1);
    }
});

//分区聚合
KeyedStream<Tuple2<String, Integer>, String> keyedStream =
mapStream.keyBy(new KeySelector<Tuple2<String, Integer>, String>() {
    @Override
    public String getKey(Tuple2<String, Integer> tp) throws
Exception {
        return tp.f0;
    }
});
SingleOutputStreamOperator<Tuple2<String, Integer>> sumStream =
keyedStream.sum("f1");

//调用sink
sumStream.print();

//执行
environment.execute();
}
}
}

```

2.5.3. Scala代码实现

- `import org.apache.flink.streaming.api.scala._`
- ```

/**
 * @Description :
 * @School:优极限学堂
 * @Official-Website: http://www.yjxxt.com
 * @Teacher:李毅大帝
 * @Mail:863159469@qq.com
 */
object Hello04WordCountByDataStreamUseScala {
 def main(args: Array[String]): Unit = {
 //1.创建执行环境(上下文)
 }
}

```

```

 val environment: StreamExecutionEnvironment =
StreamExecutionEnvironment.getExecutionEnvironment

 //2. 调用Source方法创建DataStream
 val lineStream: DataStream[String] =
environment.socketTextStream("localhost", 19523)

 //3. 调用Transformation(s)
 val wordStream: DataStream[String] = lineStream.flatMap(_.split(" "))
 val mapStream: DataStream[(String, Int)] = wordStream.map((_, 1))
 //分区聚合
 val keyByStream: KeyedStream[(String, Int), String] =
mapStream.keyBy(_.hashCode)
 val sumStream: DataStream[(String, Int)] = keyByStream.sum(1)

 //4. 调用sink
 sumStream.print()

 //5. 启动执行
 environment.execute()
}
}

```

## 2.6. 编程代码简化

### 2.6.1. Scala版本

- //调用Transformation(s)  
lineStream.flatMap(\_.split(" ")).map((\_, 1)).keyBy(\_.hashCode).sum(1).print()

### 2.6.2. Java版本

- //代码正确--可以推断出具体类型  
lineStream.map(line -> "yjxxt\_" + line).print();  
  
//代码错误--无法进行正确的类型推断  
lineStream.flatMap((line, collector) -> {  
 String[] words = line.split(" ");  
 for (String word : words) {  
 collector.collect(word);  
 }  
}).map(word -> Tuple2.of(word, 1)).keyBy(tuple2 ->  
tuple2.f0).sum("f1").print();

## 2.7. TypeInformation

### 2.7.1. 概念

- Flink程序所处理的流中的事件一般是对象类型。操作符接收对象输出对象。所以Flink的内部机制需要能够处理事件的类型。在网络中传输数据，或者将数据写入到状态后端、检查点和保存点中，都需要我们对数据进行序列化和反序列化。

- Flink使用了Type Information的概念来表达数据类型，这样就能针对不同的数据类型产生特定的序列化器，反序列化器和比较操作符。
- Flink也能够通过分析输入数据和输出数据来自动获取数据的类型信息以及序列化器和反序列化器。
- 在一些特定的情况下，例如匿名函数或者使用泛型的情况下，我们需要明确的提供数据的类型信息，来提高我们程序的性能。

## 2.7.2. 数据类型

- Flink支持Java和Scala提供的所有普通数据类型。最常用的数据类型可以做以下分类：
- Primitives（原始数据类型）
  - Java和Scala提供的所有原始数据类型都支持，例如Int（Java的Integer），String，Double等等。
- Java和Scala的Tuples（元组）
  - Tuple类是强类型，是一种组合数据类型，由固定数量的元素组成。
  - Flink实现的Java Tuple最多可以有25个元素，根据元素数量的不同，Tuple都被实现成了不同的类：Tuple1，Tuple2，一直到Tuple25。
  - Tuple的元素可以通过它们的public属性访问——f0，f1，f2等等。或者使用getField(int pos)方法来访问，元素下标从0开始：
- Scala的样例类和Java的POJO
  - 公有类
  - 无参数的公有构造器
  - 所有的字段都是公有的，可以通过getters和setters访问。
  - 所有字段的数据类型都必须是Flink支持的数据类型。

```

import org.apache.flink.streaming.api.datastream.DataStreamSource;
import
org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;

import java.io.Serializable;
import java.util.Objects;

/**
 * @Description :
 * @School:优极限学堂
 * @Official-website: http://www.yjxxt.com
 * @Teacher:李毅大帝
 * @Mail:863159469@qq.com
 */
public class Hello07ReturnTypeUsePojo {
 public static void main(String[] args) throws Exception {
 //获取程序运行的环境
 StreamExecutionEnvironment environment =
 StreamExecutionEnvironment.getExecutionEnvironment();
 //调用Source方法创建DataStream
 DataStreamSource<String> lineStream =
 environment.socketTextStream("localhost", 19523);

 //转成User对象
 lineStream.map(word -> new User(word, "123456")).print();
 }
}

```

```

 environment.execute();
 }

}

class User implements Serializable {
 private String uname;
 private String passwd;

 //此处需要生成 [无参构造器][getter和setter][equals和hashCode][toString]
}

```

### 2.7.3. 返回类型

- 由于JVM运行时候会擦除类型（泛型类型），Flink无法准确的获取到数据类型。因此，在使用Java API的时候，我们需要手工指定类型。
- 使用Scala的时候无需指定。
- 需要使用 `SingleOutputStreamOperator` 的 `returns` 方法来指定算子的返回数据类型。
  - `returns(Class<T> typeClass)`: 使用 `Class` 的方式指定返回数据类型。
  - `returns(TypeHint<T> typeHint)`: 使用 `TypeHint` 方式指定返回数据类型，通常泛型类型需要使用 `TypeHint` 来指定。
  - `returns(TypeInformation<T> typeInfo)`: 使用 `TypeInformation` 指定。

- TypeInformation
  - `TypeInformation` 是 Flink 类型系统的核心，是生成序列化/反序列化工具和 `Comparator` 的工具类。同时它还是连接 `schema` 和编程语言内部类型系统的桥梁。
  - 可以使用 `of` 方法创建 `TypeInformation`：
    - `of(Class typeClass)`: 从 `class` 创建。
    - `of(TypeHint typeHint)`: 从 `TypeHint` 创建。
- TypeHint
  - 由于泛型类型在运行时会被JVM擦除，所以说我们无法使用 `TypeInformation.of(xxx.class)` 方式指定带有泛型的类型。
  - 为了可以支持泛型类型，Flink引入了 `TypeHint`。例如我们需要获取 `Tuple2<String, Long>` 的类型信息，可以使用如下方式：
 

```

TypeInformation<Tuple2<String, Long>> info = TypeInformation.of(new
TypeHint<Tuple2<String, Long>>());
// 或者
TypeInformation<Tuple2<String, Long>> info = new
TypeHint<Tuple2<String, Long>>().getTypeInfo();
```
- Types
  - 在Flink中经常使用的类型已经预定义在了 `Types` 中。它们的 `serializer/deserializer` 和 `Comparator` 已经定义好了。
  - `Tuple` 类型既可以使用 `TypeHint` 指定又可以使用 `Types` 指定。例如 `Tuple2<String, Integer>` 类型我们可以使用如下

- TypeInformation.of(new TypeHint<Tuple2<String, Integer>>() {})
 // 或者
 Types.TUPLE(Types.STRING, Types.INT)

- 代码实现

```

import org.apache.flink.api.common.typeinfo.TypeHint;
import org.apache.flink.api.common.typeinfo.TypeInformation;
import org.apache.flink.api.common.typeinfo.Types;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.streaming.api.datastream.DataStreamSource;
import
org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;

/**
 * @Description :
 * @School:优极限学堂
 * @official-website: http://www.yjxxt.com
 * @Teacher:李毅大帝
 * @Mail:863159469@qq.com
 */
public class Hello06DataStreamCodeAbbreviation {
 public static void main(String[] args) throws Exception {
 //获取程序运行的环境
 StreamExecutionEnvironment environment =
 StreamExecutionEnvironment.getExecutionEnvironment();
 //调用Source方法创建DataStream
 DataStreamSource<String> lineStream =
 environment.socketTextStream("localhost", 19523);

 //代码正确--可以推断出具体类型
 // lineStream.map(line -> "yjxxt_" + line).print();

 //Types
 lineStream.map(w -> new Tuple2(w,
 1)).returns(Types.TUPLE(Types.STRING, Types.INT)).print("Types-");
 //TypeHint
 TypeInformation<Tuple2<String, Integer>> typeInfo1 = new
 TypeHint<Tuple2<String, Integer>>() {
 }.getTypeInfo();
 lineStream.<Tuple2<String, Integer>>map(w -> new Tuple2(w,
 1)).returns(typeInfo1).print("TypeHint-");
 //TypeInformation
 TypeInformation<Tuple2<String, Integer>> typeInfo2 =
 TypeInformation.of(new TypeHint<Tuple2<String, Integer>>() {
 });
 lineStream.<Tuple2<String, Integer>>map(w -> new Tuple2(w,
 1)).returns(typeInfo2).print("TypeInformation-");

 //代码正确--需要添加泛型和返回类型
 lineStream.<String>flatMap((line, collector) -> {
 String[] words = line.split(" ");
 for (String word : words) {
 collector.collect(word);
 }
 }).returns(Types.STRING)
 }
}

```

```

 .map(word -> Tuple2.of(word,
1)).returns(Types.TUPLE(Types.STRING, Types.INT))
 .keyBy(tuple2 -> tuple2.f0)
 .sum("f1").print();

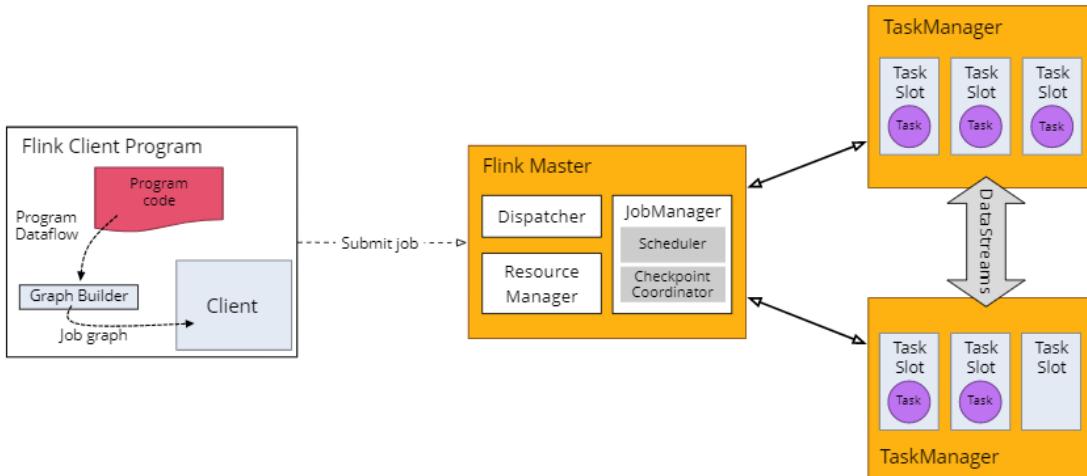
 //执行
environment.execute();
}
}

```

## 2.8. 基本概念介绍

### 2.8.1. Stream执行环境

- 每个 Flink 应用都需要有执行环境，在该示例中为 `environment`。流式应用需要用到 `StreamExecutionEnvironment`。
- DataStream API 将你的应用构建为一个 job graph，并附加到 `streamExecutionEnvironment`。
- 当调用 `environment.execute()` 时此 graph 就被打包并发送到 JobManager 上，后者对作业并行处理并将其子任务分发给 Task Manager 来执行。
- 每个作业的多个并行子任务将在 task slot 中执行。
- 注意，如果没有调用 `execute()`，应用就不会运行。
- 



- Flink的运行时环境由两个进程组成:
  - **JobManagers :**
    - JobManager有时也叫Masters，主要是协调分布式运行。他们调度任务，协调 checkpoint，协调失败任务的恢复等等
    - 一个Flink集群中至少有一台JobManager节点。高可用性的集群中将会有多个 JobManager节点，其中有一台是leader节点，其他的是备节点(standby)。
  - **TaskManagers:**
    - TaskManagers有时也叫Workers，TaskManager主要是执行dataflow中的任务 (tasks)，缓存数据以及进行数据流的交换。
    - TaskManager在同一个JVM中以多线程的方式执行任务
    - TaskManager提供了一定数量的处理插槽 (processing slots)，用于控制可以并行执行的任务数。
    - 每一个集群中至少有一个TaskManager。

- 一个TaskManager可以同时执行多个任务 (tasks)
  - 这些任务可以是同一个算子的子任务 (数据并行)
  - 这些任务可以是来自不同算子 (任务并行)
  - 这些任务可以是另一个不同应用程序 (作业并行)

## 2.8.2. 任务的执行计划

- 获取任务执行的Json串

```

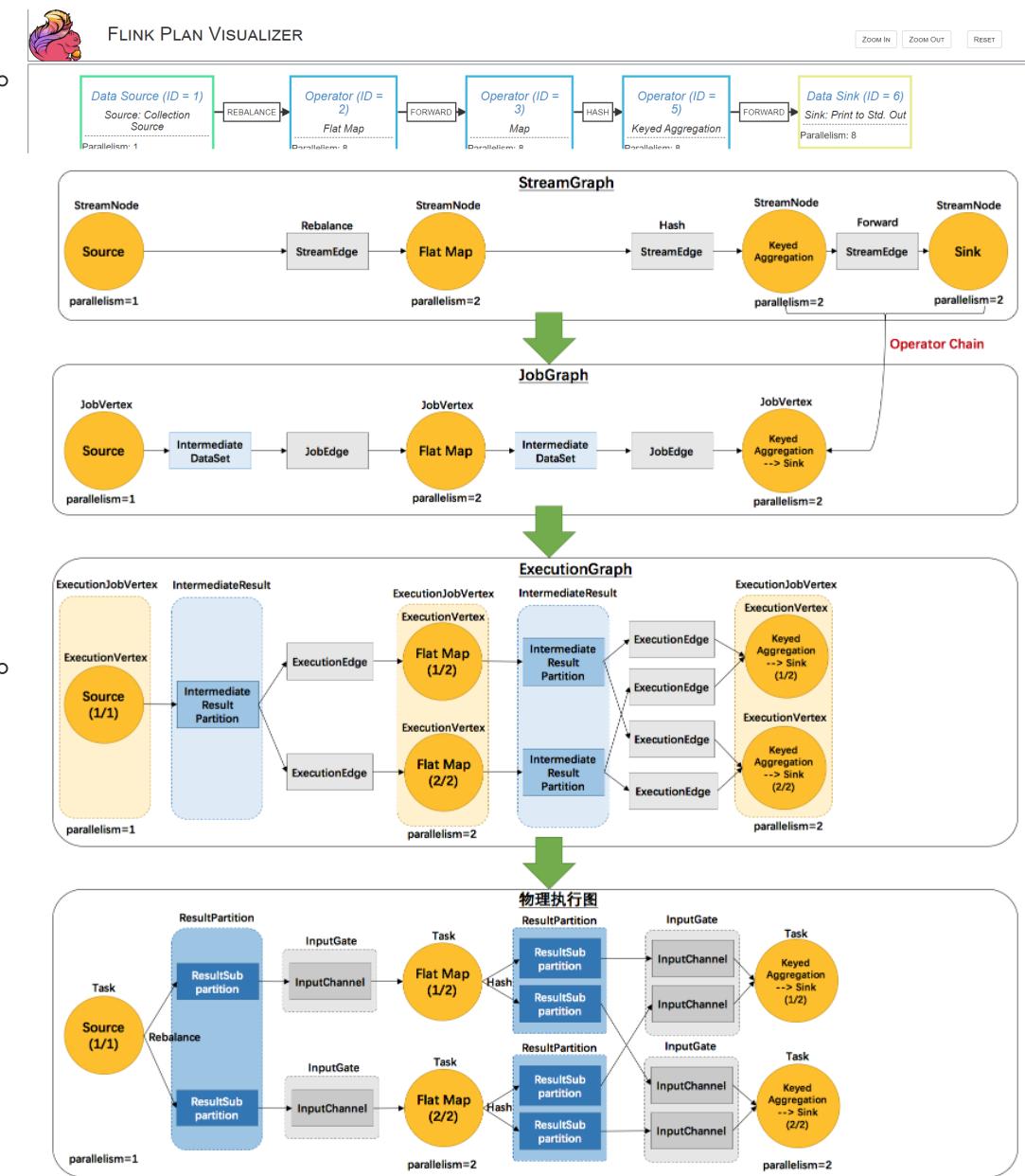
○ import org.apache.flink.api.common.typeinfo.Types;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.configuration.Configuration;
import org.apache.flink.streaming.api.datastream.DataStreamSource;
import
org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;

import java.util.Arrays;

/**
 * @Description :
 * @School:优极限学堂
 * @Official-website: http://www.yjxxt.com
 * @Teacher:李毅大帝
 * @Mail:863159469@qq.com
 */
public class Hello08ExecutionPlan {
 public static void main(String[] args) throws Exception {
 //获取程序运行的环境
 StreamExecutionEnvironment environment =
StreamExecutionEnvironment.createLocalEnvironmentWithWebUI(new
Configuration());
 //调用Source方法创建DataStream
 DataStreamSource<String> lineStream =
environment.fromElements("hello world", "hello moto");
 //开始进行计算
 lineStream.<String>flatMap((line, collector) ->
Arrays.stream(line.split(
""))
.forEach(collector::collect))
.returns(Types.STRING)
.map(word -> new Tuple2(word,
1))
.returns(Types.TUPLE(Types.STRING, Types.INT))
.keyBy(tuple2 -> tuple2.f0)
.sum(1)
.print();
 //获取执行计划
 System.out.println(environment.getExecutionPlan());
 //执行
 // environment.execute();
 }
}

```

- 查看执行图
  - <http://flink.apache.org/visualizer/>



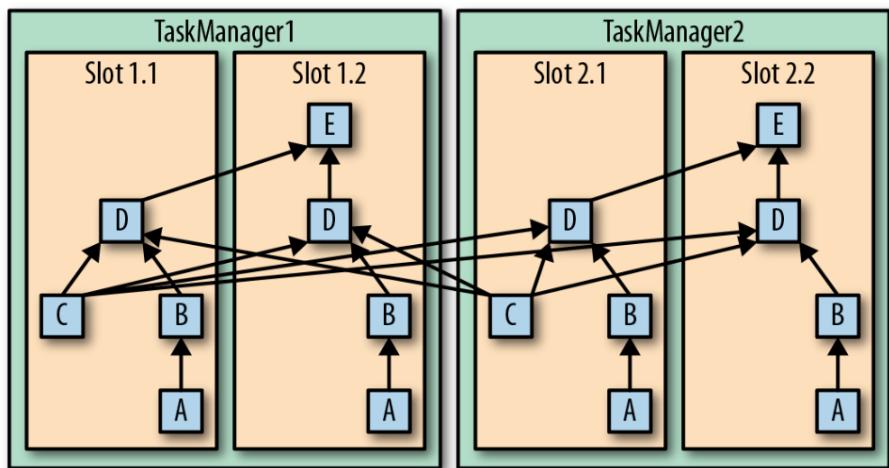
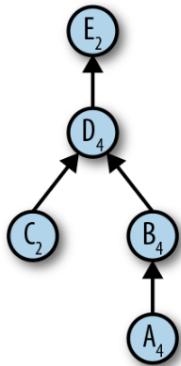
### • 图形解释

- Flink 中的执行图可以分成四层: StreamGraph -> JobGraph -> ExecutionGraph -> 物理执行图。
- **StreamGraph:**
  - 根据用户通过 Stream API 编写的代码生成的最初的图。用来表示程序的拓扑结构。
- **JobGraph:**
  - StreamGraph 经过优化后生成了 JobGraph，提交给 JobManager 的数据结构。主要的优化为，将多个符合条件的节点 chain 在一起作为一个节点，这样可以减少数据在节点之间流动所需要的序列化/反序列化/传输消耗。
- **ExecutionGraph:**
  - JobManager 根据 JobGraph 生成 ExecutionGraph。
  - ExecutionGraph 是 JobGraph 的并行化版本，是调度层最核心的数据结构。
- **物理执行图:**
  - JobManager 根据 ExecutionGraph 对 Job 进行调度后，在各个 TaskManager 上部署 Task 后形成的“图”
  - 并不是一个具体的数据结构。

### 2.8.3. 任务并行度

- 一个算子的子任务(subtask)的个数被称之为并行度(parallelism ['pærəlelɪzəm] )
- 一个程序中，不同的算子可能具有不同的并行度。

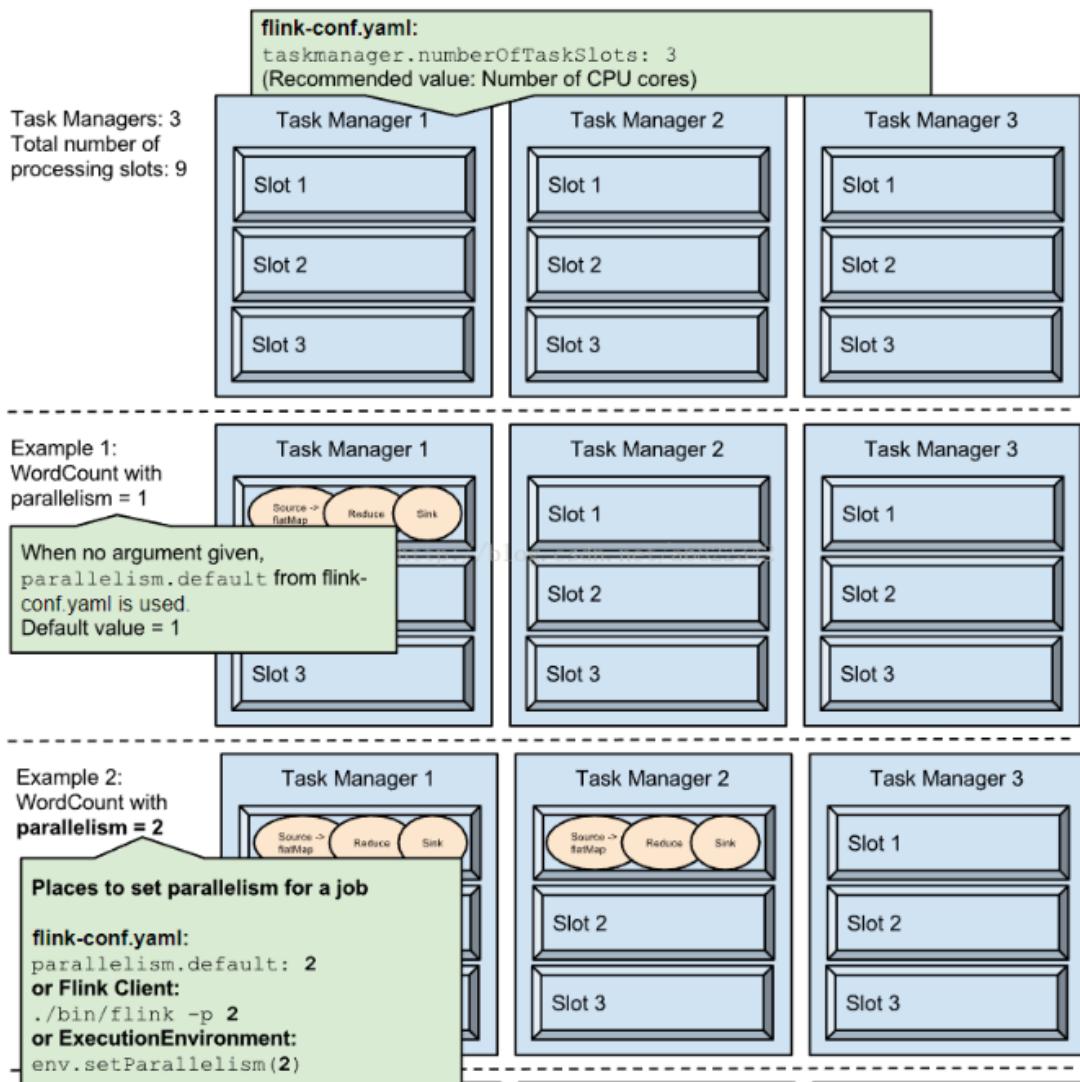
- JobGraph



本次Job包含了5个算子。算子A和C是数据源 (source) , E是输出端 (sink) 。C和E并行度为2, 而其他的算子并行度为4。因为最高的并行度是4, 所以应用需要至少四个slot来执行任务。

现在有两个TaskManager, 每个又各有两个slot, 所以我们的需求是满足的。作业管理器将JobGraph转化为“执行图” (ExecutionGraph), 并将任务分配到四个可用的slot上。**对于有4个并行任务的算子, 它的task会分配到每个slot上**。而对于并行度为2的operator C和E, 它们的任务被分配到slot 1.1、2.1以及 slot 1.2、2.2。将tasks调度到slots上, 可以让多个tasks跑在同一个TaskManager内, 也可以是的tasks之间的数据交换更高效。

- 



- 将 slot 的个数配置为 3 (taskmanager.numberOfTaskSlots) ,每个TaskManager会分配 3 个 Slot 来执行 task, 如果配置了 3 个 taskmanager 那么就如图一共有 9 个 Slot。
- Parallelism是指 TaskManager 在实际运行过程中的并发。默认并行度的配置为 1 (parallelism.default) , 那么如图 9 个 Slot 只有一个是在工作的, 其他 8 个都空闲。

## 2.8.4. 并行度设置

- Flink程序的任务并行度设置分为四个级别。
- 配置文件
  - 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。
- 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。
- 算子级别
  - 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。
- 代码实现

```

○ import org.apache.flink.api.common.typeinfo.Types;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.streaming.api.datastream.DataStreamSource;
import
org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;

/**
 * @Description :
 * @School:优极限学堂
 * @Official-website: http://www.yjxxt.com
 * @Teacher:李毅大帝
 * @Mail:863159469@qq.com
 */
public class Hello09FlinkParallenism {
 public static void main(String[] args) throws Exception {
 //获取程序运行的环境
 StreamExecutionEnvironment environment =
 StreamExecutionEnvironment.getExecutionEnvironment();
 environment.setParallelism(2);
 //调用Source方法创建DataStream
 DataStreamSource<String> lineStream =
 environment.socketTextStream("localhost", 19523).setParallelism(1);

 //代码正确--需要添加泛型和返回类型
 lineStream
 .map(w -> new Tuple2(w,
 1)).returns(Types.TUPLE(Types.STRING, Types.INT))
 .print("Types-").setParallelism(1);
 //执行
 environment.execute();
 }
}

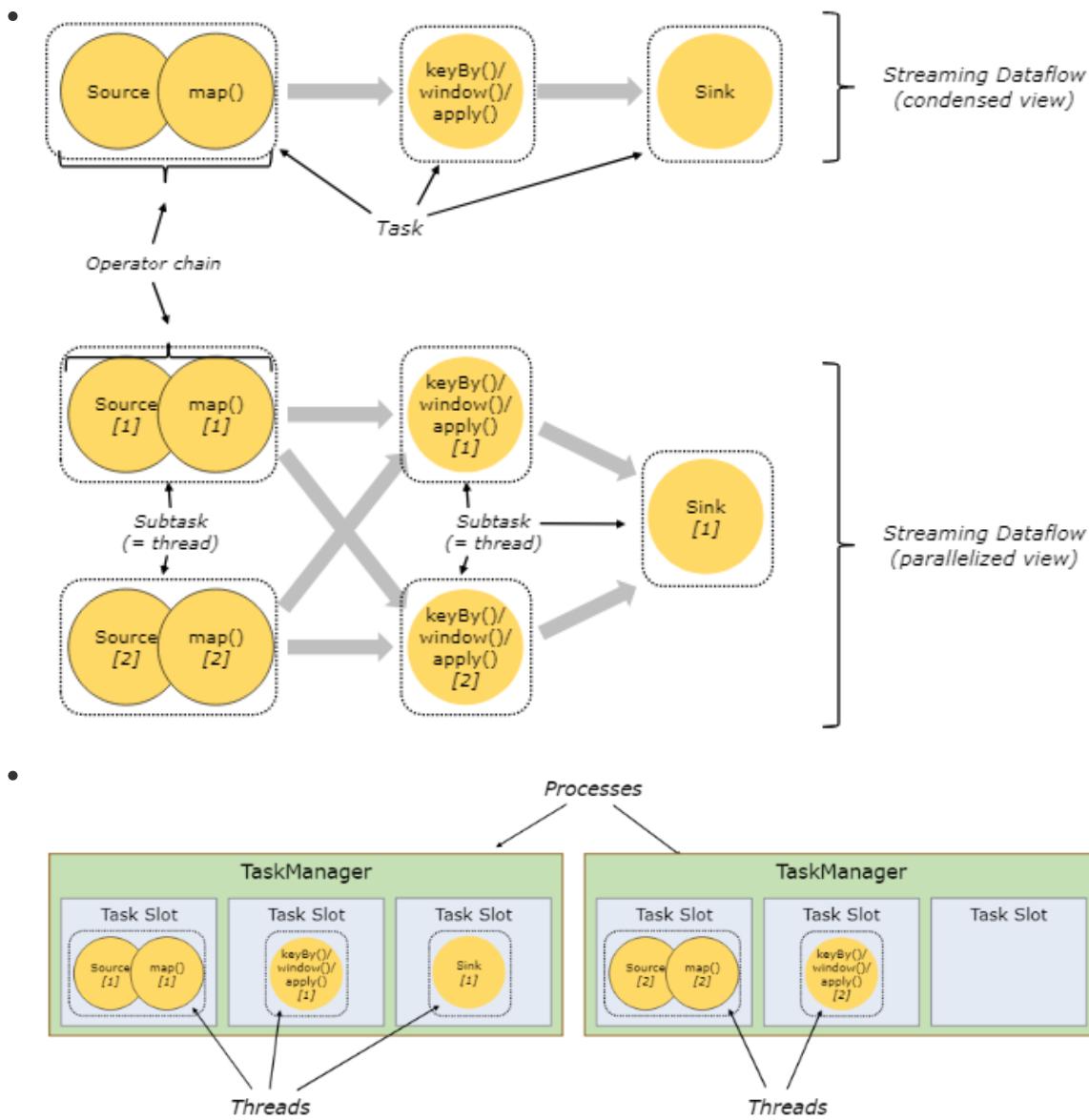
```

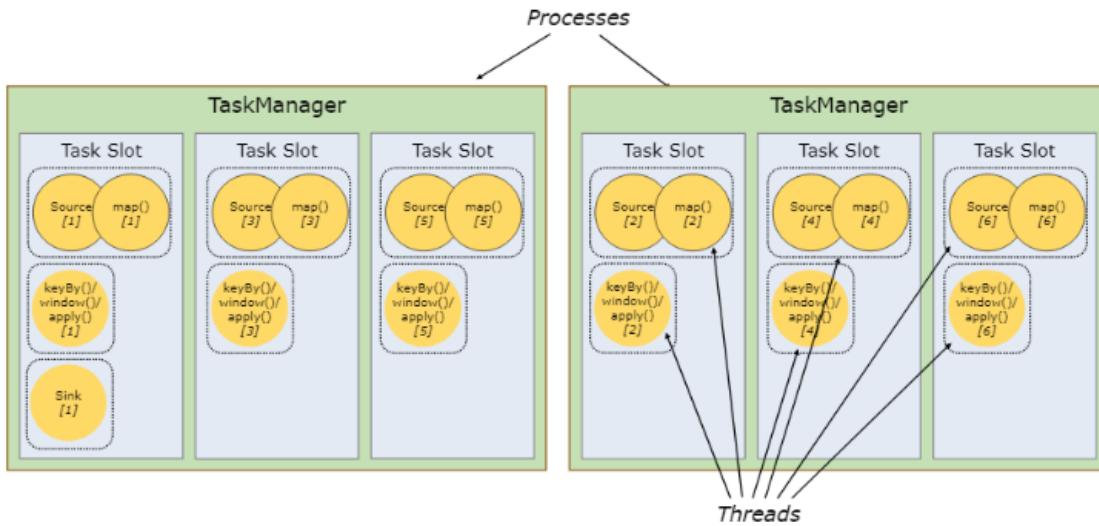
## 2.8.5. Flink操作链

- 在分布式环境下，Flink将操作的子任务链在一起组成一个任务，每一个任务在一个线程中执行。将操作链在一起是一个不错的优化：它减少了线程间的切换和缓冲，提升了吞吐量同时减低了时延。
- 操作链条件
  - 上下游算子实例间是 oneToOne 数据传输（forward）；
  - 上下游算子并行度相同；
  - 上下游算子属于相同的 slotSharingGroup（槽位共享组）；
- 链接的行为可以在编程API中进行指定开启操作链（默认）和禁用操作链的

```
○ // 当前环境关闭操作链..
environment.disableOperatorChaining();
```

```
//单个算子关闭操作链
Stream.disableChaining()
```





## 3. Flink 运行环境

### 3.1. 批处理运行环境

- `ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();`

### 3.2. 流式计算运行环境

- `StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();`

### 3.3. 本地web ui环境

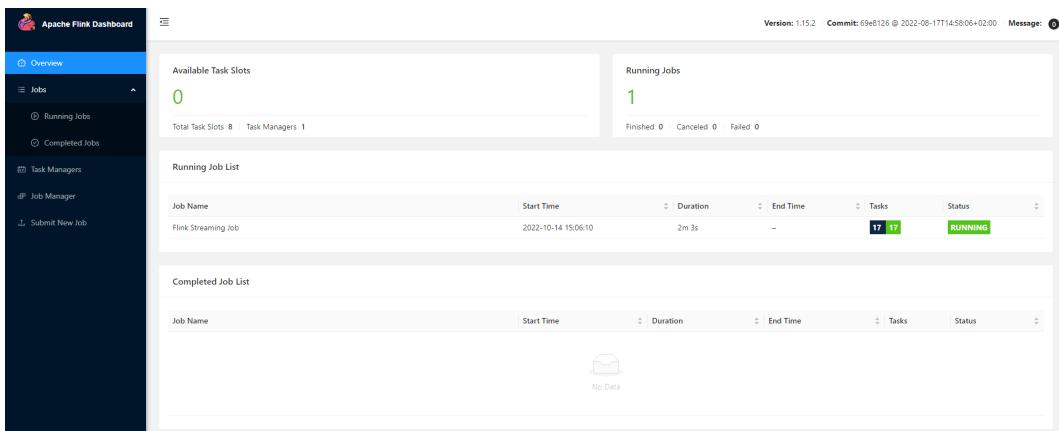
- 添加pom依赖

- ```
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-runtime-web_2.12</artifactId>
    <version>${flink.version}</version>
</dependency>
```

- 代码实现

- ```
//默认访问的端口为: 8081
StreamExecutionEnvironment environment =
StreamExecutionEnvironment.createLocalEnvironmentWithWebUI(new
Configuration());
```

-



## 4. Flink Source类算子

通过 `StreamExecutionEnvironment` 可以访问多种预定义的 stream source：

### 4.1. 基于文件：

- `readTextFile(path)` - 读取文本文件，例如遵守 TextInputFormat 规范的文件，逐行读取并将它们作为字符串返回。
- `readFile(fileInputFormat, path)` - 按照指定的文件输入格式读取（一次）文件。
- `readFile(fileInputFormat, path, watchType, interval, pathFilter, typeInfo)` - 这是前两个方法内部调用的方法。

```
○ DataSet<String> source =
environment.readTextFile("data/wordcount.txt");
```

### 4.2. 基于套接字：

- `socketTextStream` - 从套接字读取。元素可以由分隔符分隔。
  - 在启动 Flink 程序之前，必须先启动一个 Socket 服务
  - `DataStreamSource<String> lineStream =
environment.socketTextStream("localhost", 19523);`

### 4.3. 基于集合：

- `fromCollection(collection)` - 从 Java Java.util.Collection 创建数据流。集合中的所有元素必须属于同一类型。
- `fromCollection(Iterator, Class)` - 从迭代器创建数据流。class 参数指定迭代器返回元素的数据类型。
- `fromElements(T ...)` - 从给定的对象序列中创建数据流。所有的对象必须属于同一类型。
- `fromParallelCollection(SplittableIterator, Class)` - 从迭代器并行创建数据流。class 参数指定迭代器返回元素的数据类型。
- `generateSequence(from, to)` - 基于给定间隔内的数字序列并行生成数据流。

- ```

import org.apache.flink.streaming.api.datastream.DataStreamSource;
import
org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;

import java.util.List;

/**
 * @Description :
 * @School:优极限学堂
 * @Official-website: http://www.yjxxt.com
 * @Teacher:李毅大帝
 * @Mail:863159469@qq.com
 */
public class Hello01SourceFromCollection {
    public static void main(String[] args) throws Exception {
        //获取程序运行的环境
        StreamExecutionEnvironment environment =
        StreamExecutionEnvironment.getExecutionEnvironment();
        //通过集合获取数据源
        List<String> list = List.of("11", "22", "33", "44", "55", "66",
        "77");
        DataStreamSource<String> collectionSource =
        environment.fromCollection(list);
        collectionSource.map(word -> "collectionSource-" + word).print();

        //通过元素获取数据源
        DataStreamSource<String> elementSource =
        environment.fromElements("aa", "bb", "cc", "dd", "ee", "ff");
        elementSource.map(word -> "elementSource-" + word).print();

        //自动生成数据源
        DataStreamSource<Long> sequenceSource =
        environment.generateSequence(1, 5);
        sequenceSource.map(word -> "SequenceSource-" + word).print();

        environment.execute();
    }
}

```

4.4. 基于Connectors:

- 一些比较基本的 Source 和 Sink 已经内置在 Flink 里。
 - 预定义 data sources 支持从文件、目录、socket，以及 collections 和 iterators 中读取数据。
 - 预定义 data sinks 支持把数据写入文件、标准输出 (stdout) 、标准错误输出 (stderr) 和 socket。

连接器可以和多种多样的第三方系统进行交互。目前支持以下系统：

- Apache Kafka (source/sink)
- Apache Cassandra (sink)
- Amazon Kinesis Streams (source/sink)
- Elasticsearch (sink)
- FileSystem (sink)
- RabbitMQ (source/sink)
- Google PubSub (source/sink)
- Hybrid Source (source)
- Apache NiFi (source/sink)
- Apache Pulsar (source)
- JDBC (sink)

请记住，在使用一种连接器时，通常需要额外的第三方组件，比如：数据存储服务器或者消息队列。要注意这些列举的连接器是 Flink 工程的一部分，包含在发布的源码中，但是不包含在二进制发行版中。更多说明可以参考对应的子部分。

- Apache Kafka 连接器

- Flink 提供了 Apache Kafka 连接器使用精确一次 (Exactly-once) 的语义在 Kafka topic 中读取和写入数据。
- 当前 Kafka client 向后兼容 0.10.0 或更高版本的 Kafka broker。
- 依赖

- ```
<!-- Apache Kafka Connectors -->
<dependency>
 <groupId>org.apache.flink</groupId>
 <artifactId>flink-connector-kafka</artifactId>
 <version>1.15.2</version>
</dependency>
<dependency>
 <groupId>org.apache.flink</groupId>
 <artifactId>flink-connector-base</artifactId>
 <version>1.15.2</version>
</dependency>
```

- 代码实现

- ```
import com.yjxxt.util.KafkaUtil;
import org.apache.flink.api.common.eventtime.WatermarkStrategy;
import
org.apache.flink.api.common.serialization.SimpleStringSchema;
import org.apache.flink.connector.kafka.source.KafkaSource;
import
org.apache.flink.connector.kafka.source.enumerator.initializer.OffsetsInitializer;
import org.apache.flink.streaming.api.datastream.DataStreamSource;
import
org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;

import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;

/**
 * @Description :
 * @School:优极限学堂
 * @Official-Website: http://www.yjxxt.com
 * @Teacher:李毅大帝
 * @Mail:863159469@qq.com
 */
public class Hello02SourcekafkaConnector {
    public static void main(String[] args) throws Exception {
```

```

        //启动一个线程专门发送消息给Kafka，这样我们才有数据消费
        new Thread(() -> {
            for (int i = 0; i < 100; i++) {
                KafkaUtil.sendMsg("yjxxt",
                LocalDateTime.now().format(DateTimeFormatter.ISO_DATE_TIME));
            }
        }).start();

        //获取环境
        StreamExecutionEnvironment environment =
        StreamExecutionEnvironment.getExecutionEnvironment();

        //设置Kafka连接
        KafkaSource<String> source = KafkaSource.<String>builder()

        .setBootstrapServers("node01:9092,node02:9092,node03:9092")
        .setTopics("yjxxt")
        .setGroupId("flink_KafkaConnector")
        .setStartingOffsets(OffsetsInitializer.earliest())
        .setValueOnlyDeserializer(new SimpleStringSchema())
        .build();

        //读取数据源
        DataStreamSource<String> kafkaSource =
        environment.fromSource(source, WatermarkStrategy.noWatermarks(),
        "Kafka Source");
        kafkaSource.map(word -> "kafkaSource_" + word).print();

        //执行环境
        environment.execute();
    }
}

```

4.5. 自定义Source:

- Flink 的 DataStream API 可以让开发者根据实际需要， 灵活的自定义 Source。
- 本质上就是定义一个类，
 - 可以实现 SourceFunction 或者 RichSourceFunction , 这两者都是非并行的 source 算子
 - 也可实现 ParallelSourceFunction 或者 RichParallelSourceFunction , 这两者都是可并行执行的 source 算子
 - 带 Rich 的， 都拥有 open() ,close() ,getRuntimeContext() 方法
 - 带 Parallel 的， 都可多实例并行执行
- 要解析的数据

- 7V6SQAAfMvLn+ztqouIObQBBeI395kedzcow8tH6NW13YPeyPsDxvPBA/Ob7nsiwyNj0po
i2cI=
g5sr0Jt7zpHRojrwNC9oa5cFnDoGwOFXwr0NqKu0I2X+dIIiqDWrvU2H6IfQFBVQwAooNTeH
5ako=
XpH58ZN4zd0LZ+5rTHyNhQMkBTVUPSAzA02ot4x5fvsqmhPrHwk1g3QJ/2RwNqpwkGtFUOX
kJc4cv50wGjwozg==
r5msovDdEd6LezYrKopsoL75wfV81/76v0+TsQ6jw2CwYafcIKiNrTLFA92MjFj1CYqAzq1
0SJxYS30qB2iqV+xQ035wjKM+QZ7M4qRxxWc=

- 代码实现【基础版】

```
○ import com.yjxxt.util.DESUtil;
import org.apache.commons.io.FileUtils;
import org.apache.flink.streaming.api.datastream.DataStreamSource;
import
org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import
org.apache.flink.streaming.api.functions.source.ParallelSourceFunction;

import java.io.File;
import java.util.List;

/**
 * @Description :
 * @School:优极限学堂
 * @Official-website: http://www.yjxxt.com
 * @Teacher:李毅大帝
 * @Mail:863159469@qq.com
 */
public class Hello03SourceCustom {
    public static void main(String[] args) throws Exception {
        //获取环境
        StreamExecutionEnvironment environment =
        StreamExecutionEnvironment.getExecutionEnvironment();

        //读取数据源
        DataStreamSource<String> customImplSource =
environment.addSource(new YjxxtCustomSource()).setParallelism(2);
        customImplSource.map(word -> "customImplSource-" +
word).print().setParallelism(1);

        //执行环境
        environment.execute();
    }
}

/**
 * 实现接口，只有run和cancel可以被重写
 */
class YjxxtCustomSource implements ParallelSourceFunction<String> {

    /**
     * @param context
     * @throws Exception
     */
    @Override
    public void run(SourceContext<String> context) throws Exception {
```

```

        //开始读取文件
        List<String> lines = FileUtils.readLines(new
File("data/secret.txt"), "utf-8");
        //开始进行遍历并解密
        for (String line : lines) {
            //开始解密
            context.collect(DESUtil.decrypt("yjxxt0523", line));
        }
    }

    @Override
    public void cancel() {
        System.out.println("YjxxtCustomSource.cancel_" +
System.currentTimeMillis());
    }
}

```

- 代码实现【Rich版】

```

o import com.yjxxt.util.DESUtil;
import org.apache.commons.io.FileUtils;
import org.apache.flink.configuration.Configuration;
import org.apache.flink.streaming.api.datastream.DataStreamSource;
import
org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import
org.apache.flink.streaming.api.functions.source.RichParallelSourceFunct
ion;

import java.io.File;
import java.util.List;

/**
 * @Description :
 * @School:优极限学堂
 * @Official-website: http://www.yjxxt.com
 * @Teacher:李毅大帝
 * @Mail:863159469@qq.com
 */
public class Hello04SourceCustomRich {
    public static void main(String[] args) throws Exception {
        //获取环境
        StreamExecutionEnvironment environment =
StreamExecutionEnvironment.getExecutionEnvironment();

        //读取数据源
        DataStreamSource<String> customExtSource =
environment.addSource(new YjxxtCustomSourceRich()).setParallelism(2);
        customExtSource.map(word -> "customExtSource-" +
word).print().setParallelism(1);
        //执行环境
        environment.execute();
    }
}

/**
 * 比较富有，除了run和cancel可以被重写

```

```

* 还有open、close方法可以被重写，最主要的是可以获取运行时的状态RuntimeContext
*/
class YjxxtCustomSourceRich extends RichParallelSourceFunction<String>
{

    @Override
    public void open(Configuration parameters) throws Exception {
        System.out.println("YjxxtCustomSourceExt.open" +
System.currentTimeMillis());
        super.open(parameters);
    }

    @Override
    public void close() throws Exception {
        System.out.println("YjxxtCustomSourceExt.close" +
System.currentTimeMillis());
        super.close();
    }

    /**
     * @param context
     * @throws Exception
     */
    @Override
    public void run(SourceContext<String> context) throws Exception {
        //开始读取文件
        List<String> lines = FileUtils.readLines(new
File("data/secret.txt"), "utf-8");
        //Task总数
        int taskCount =
this.getRuntimeContext().getNumberOfParallelSubtasks();
        //当前TaskID
        int taskId = this.getRuntimeContext().getIndexofThisSubtask();

        //开始进行遍历并解密
        for (String line : lines) {
            //如果line解密后取余taskCount的结果等于taskId，就由当前线程去接受
            String decrypt = DESUtil.decrypt("yjxxt0523", line);
            if (decrypt.hashCode() % taskCount == taskId) {
                //开始解密
                context.collect(taskId + ":" + decrypt);
            }
        }
    }

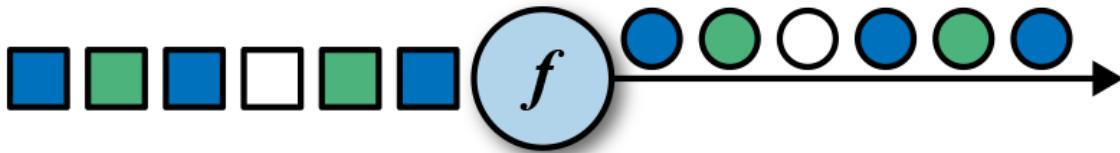
    @Override
    public void cancel() {
        System.out.println("YjxxtCustomSource.cancel_" +
System.currentTimeMillis());
    }
}

```

5. Flink Transformation类算子

- 用户通过算子能将一个或多个 DataStream 转换成新的 DataStream，在应用程序中可以将多个数据转换算子合并成一个复杂的数据流拓扑。
- 这部分内容将描述 Flink DataStream API 中基本的数据转换 API，数据转换后各种数据分区方式，以及算子的链接策略。

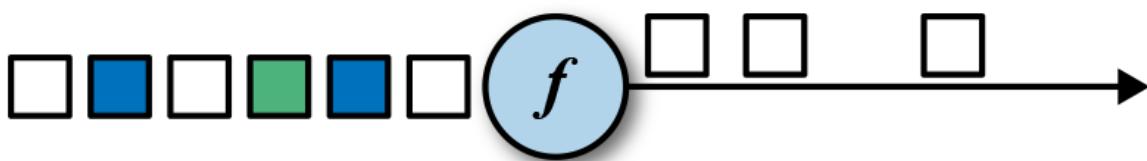
5.1. map



- 输入一个元素同时输出一个元素。下面是将输入流中元素数值加倍的 map function：

```
• DataStreamSource<Integer> mapSource = environment.fromElements(1, 2, 3, 4,
    5, 6, 7, 8, 9);
mapSource.map(new MapFunction<Integer, String>() {
    @Override
    public String map(Integer integer) throws Exception {
        return "yjxxt_" + integer;
    }
}).print();
```

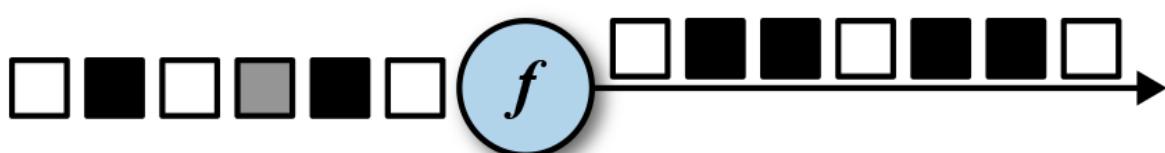
5.2. filter



- 为每个元素执行一个布尔 function，并保留那些 function 输出值为 true 的元素。

```
• DataStreamSource<Integer> filterSource = environment.fromElements(1, 2, 3,
    4, 5, 6, 7, 8, 9);
filterSource.filter(new FilterFunction<Integer>() {
    @Override
    public boolean filter(Integer integer) throws Exception {
        return integer % 2 == 0;
    }
}).print();
```

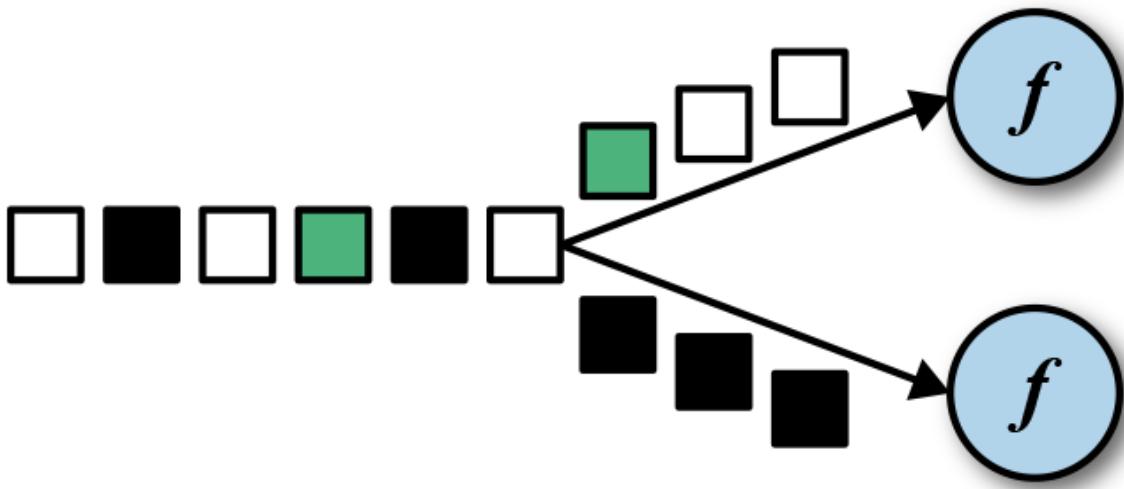
5.3. flatMap



- 输入一个元素同时产生零个、一个或多个元素。

- ```
• DataStreamSource<String> flatmapSource = environment.fromElements("hello world", "hello ketty", "hello pretty");
flatmapSource.flatMap(new FlatMapFunction<String, String>() {
 @Override
 public void flatMap(String str, Collector<String> collector) throws
Exception {
 String[] words = str.split(" ");
 for (String word : words) {
 collector.collect(word);
 }
}
}).print();
```

## 5.4. keyby



- 在逻辑上将流划分为不相交的分区。具有相同 key 的记录都分配到同一个分区。在内部，`keyBy()`是通过哈希分区实现的。

- ```
• DataStreamSource<String> keybySource = environment.fromElements("a", "bb",
"ccc", "aaa", "dd");
keybySource.keyBy(new KeySelector<String, Integer>() {
    @Override
    public Integer getKey(String s) throws Exception {
        return s.length();
    }
}).print();
```

5.5. aggregation

- 滚动聚合算子由 `KeyedStream` 调用，并生成一个聚合以后的 `DataStream`
- 滚动聚合算子是多个聚合算子的统称，有 `sum`、`min`、`minBy`、`max`、`maxBy`；
- 滚动聚合方法：
 - `sum()`: 在输入流上对指定的字段做滚动相加操作。
 - `min()`: 在输入流上对指定的字段求最小值。

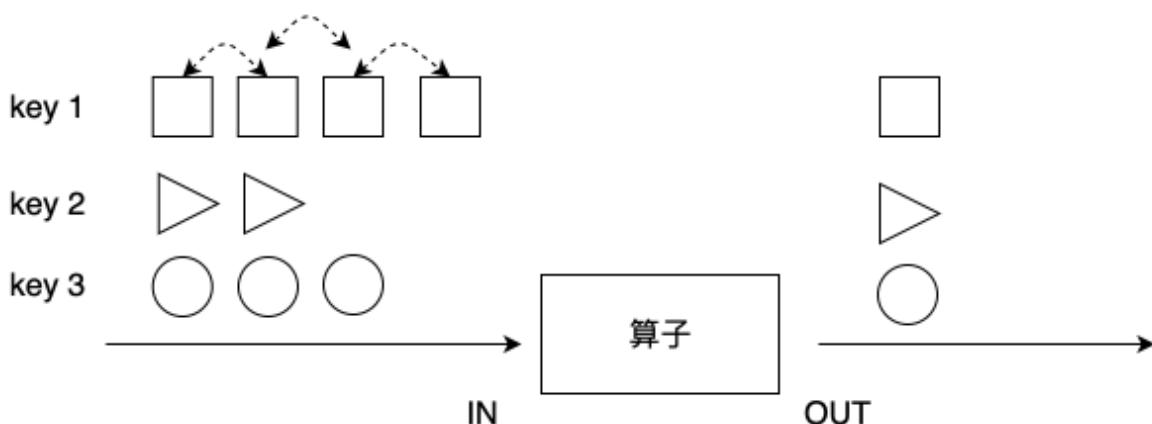
- max(): 在输入流上对指定的字段求最大值。
- minBy(): 在输入流上针对指定字段求最小值，并返回最小值字段所在的那条数据。
- maxBy(): 在输入流上针对指定字段求最大值，并返回最大值字段所在的那条数据。

```

• List<Tuple2<String, Integer>> list = new ArrayList<>();
list.add(new Tuple2<>("math2", 200));
list.add(new Tuple2<>("chinese2", 20));
list.add(new Tuple2<>("math1", 100));
list.add(new Tuple2<>("chinese1", 10));
list.add(new Tuple2<>("math4", 400));
list.add(new Tuple2<>("chinese4", 40));
list.add(new Tuple2<>("math3", 300));
list.add(new Tuple2<>("chinese3", 30));
DataStreamSource<Tuple2<String, Integer>> aggregationSource =
environment.fromCollection(list);
KeyedStream<Tuple2<String, Integer>, Integer> keyedStream =
aggregationSource.keyBy(new KeySelector<Tuple2<String, Integer>, Integer>()
{
    @Override
    public Integer getKey(Tuple2<String, Integer> tuple2) throws Exception {
        return tuple2.f0.length();
    }
});
keyedStream.sum(1).print("sum-");
keyedStream.max(1).print("max-");
keyedStream.maxBy(1).print("maxBy-");
keyedStream.min(1).print("min-");
keyedStream.minBy(1).print("minBy-");

```

5.6. reduce



- 在相同 key 的数据流上“滚动”执行 reduce。将当前元素与最后一次 reduce 得到的值组合然后输出新值。

```

• import org.apache.flink.api.common.typeinfo.Types;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.streaming.api.datastream.DataStreamSource;
import
org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;

import java.util.ArrayList;
import java.util.Arrays;

```

```

import java.util.List;

/**
 * @Description :
 * @School:优极限学堂
 * @Official-website: http://www.yjxxt.com
 * @Teacher:李毅大帝
 * @Mail:863159469@qq.com
 */
public class Hello05TransformationCommon {
    public static void main(String[] args) throws Exception {
        //获取环境
        StreamExecutionEnvironment environment =
        StreamExecutionEnvironment.getExecutionEnvironment();

        //获取数据源
        List<String> list = new ArrayList<>();
        list.add("hello,hadoop");
        list.add("hello,hbase");
        list.add("hello,,,moto");
        list.add("hello,hive");
        DataStreamSource<String> lineStream =
        environment.fromCollection(list);
        //常见操作
        lineStream.flatMap((line, collector) -> {

            Arrays.stream(line.split(",")).forEach(collector::collect);
            }, Types.STRING)
            .filter(word -> word.length() > 0)
            .map(word -> Tuple2.of(word, 1), Types.TUPLE(Types.STRING,
            Types.INT))
            .keyBy(tuple -> tuple.f0)
            .reduce((t1, t2) -> {
                t1.f1 = t1.f1 + t2.f1;
                return t1;
            })
            .print();
        //执行环境
        environment.execute();
    }
}

```

5.7. Iterate

- 通过将一个算子的输出重定向到某个之前的算子来在流中创建“反馈”循环。这对于定义持续更新模型的算法特别有用。

```

import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.datastream.DataStreamSource;
import org.apache.flink.streaming.api.datastream.IterativeStream;
import
org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;

/**
 * @Description :

```

```

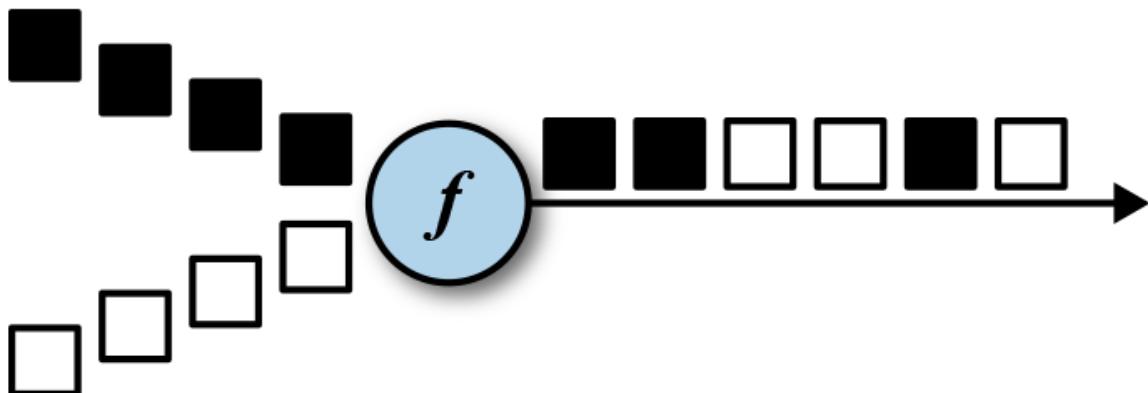
* @school:优极限学堂
* @official-website: http://www.yjxxt.com
* @Teacher:李毅大帝
* @Mail:863159469@qq.com
*/
public class Hello06TransformationIterate {
    public static void main(String[] args) throws Exception {
        //获取环境
        StreamExecutionEnvironment environment =
        StreamExecutionEnvironment.getExecutionEnvironment();
        environment.setParallelism(1);
        //获取数据源
        DataStreamSource<Integer> source = environment.fromElements(50, 66,
77);
        //进入循环
        IterativeStream<Integer> iteration = source.iterate();
        //循环体
        DataStream<Integer> iterationBody = iteration.map(num -> num - 10);
        //循环条件
        DataStream<Integer> feedback = iterationBody.filter(num -> num >
10);
        //开始迭代
        iteration.closeWith(feedback);

        //找出不满足条件的变量
        DataStream<Integer> output = iterationBody.filter(num -> num <= 10);
        output.print("不满足条件的输出: ");
        //执行环境
        environment.execute();
    }
}

```

- 问题：统计每个数字分别减少了多少次？

5.8. union



- 将两个或多个数据流联合来创建一个包含所有流中数据的新流。注意：如果一个数据流和自身进行联合，这个流中的每个数据将在合并后的流中出现两次。
- 事件合流的方式为**FIFO方式**。操作符并不会产生一个特定顺序的事件流。**union操作符也不会进行去重**。每一个输入事件都被发送到了下一个操作符。
 - 1.union 合并的流的元素必须是相同的
 - 2.union 可以合并多条流

- ```
• DataStreamSource<Integer> source1 = environment.fromElements(1, 3, 5, 7, 9, 11);
DataStreamSource<Integer> source2 = environment.fromElements(2, 4, 6, 8, 0, 11);
DataStream<Integer> unionStream = source1.union(source2);
unionStream.print();
```

## 5.9. connect

- “连接”两个数据流并保留各自的类型。connect 允许在两个流的处理逻辑之间共享状态。
- 两个DataStream经过connect之后被转化为ConnectedStreams,
  - ConnectedStreams会对两个流的数据应用不同的处理方法，且双流之间可以共享状态。
  - ConnectedStreams提供了 map() 和 flatMap() 方法，分别需要接收类型为 CoMapFunction 和 CoFlatMapFunction 的参数
- 代码实现

```
○ import org.apache.flink.streaming.api.datastream.ConnectedStreams;
import org.apache.flink.streaming.api.datastream.DataStreamSource;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.functions.co.CoMapFunction;

/**
 * @Description :
 * @School:优极限学堂
 * @Official-website: http://www.yjxxt.com
 * @Teacher:李毅大帝
 * @Mail:863159469@qq.com
 */
public class Hello07TransformationConnect {
 public static void main(String[] args) throws Exception {
 //获取程序运行的环境
 StreamExecutionEnvironment environment =
 StreamExecutionEnvironment.getExecutionEnvironment();

 //获取数据源
 DataStreamSource<Integer> source1 =
environment.fromElements(33, 35, 38, 45);
 DataStreamSource<String> source2 =
environment.fromElements("no", "no", "yes", "no");
 ConnectedStreams<Integer, String> connectedStreams =
source1.connect(source2);
 connectedStreams.map(new CoMapFunction<Integer, String, String>
() {
 @Override
 public String map1(Integer integer) throws Exception {
 if (integer > 40) {
 return "温度[" + integer + "]异常，准备报警";
 }
 return "温度[" + integer + "]正常";
 }
 });

 @Override
```

```

 public String map2(String s) throws Exception {
 if ("yes".equals(s)) {
 return "监控[" + s + "]异常，准备报警";
 }
 return "监控[" + s + "]正常";
 }
 }).print();

 environment.execute();
}
}

```

## 6. Flink Sink类算子

- Data sinks 使用 DataStream 并将它们转发到文件、套接字、外部系统或打印它们。

### 6.1. 输出到控制台

- print
  - 将计算结果打印到控制台，通常是用来做实验和测试时使用。

### 6.2. 输出到文件

- 这些方法已经被@Deprecated,请谨慎使用
- writeAsText
  - 将计算结果输出成text文件
- writeAsCsv
  - 写出的数据格式必须为Tuple，否则就会报错
  - 将计算结果输出成csv文件
- writeUsingOutputFormat
  - 自定义输出方式。
  - 尝试自己实现将一段话通过DES加密
- 代码实现

```

 import org.apache.flink.api.common.typeinfo.Types;
 import org.apache.flink.api.java.tuple.Tuple2;
 import org.apache.flink.core.fs.FileSystem;
 import org.apache.flink.streaming.api.datastream.DataStreamSource;
 import
 org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;

 /**
 * @Description :
 * @School:优极限学堂
 * @official-website: http://www.yjxxt.com
 * @Teacher:李毅大帝
 * @Mail:863159469@qq.com
 */
 public class Hello08sinkToFile {

```

```

public static void main(String[] args) throws Exception {
 //获取环境
 StreamExecutionEnvironment environment =
 StreamExecutionEnvironment.getExecutionEnvironment();
 environment.setParallelism(2);
 //获取数据源
 DataStreamSource<String> lineStream =
environment.fromElements("1", "2", "3", "4", "5", "6", "7", "8");
 //常见操作
 lineStream.map(word -> "yjxxt_" + word)
 .writeAsText("data/text_" +
System.currentTimeMillis());
 lineStream.map(word -> Tuple2.of(word, 666),
Types.TUPLE(Types.STRING, Types.INT))
 .writeAsCsv("data/csv_" + System.currentTimeMillis());

 //执行环境
 environment.execute();
}
}

```

## 6.3. 输出到服务器

- writeToSocket
  - 将计算结果输出到某台机器的端口上。

## 6.4. 基于Connectors

- KafkaSink
  - `Kafkasink` 可将数据流写入一个或多个 Kafka topic。
  - ```

package com.yjxxt.flink;

import org.apache.flink.api.common.serialization.SimpleStringSchema;
import org.apache.flink.connector.base.DeliveryGuarantee;
import
org.apache.flink.connector.kafka.sink.KafkaRecordSerializationSchema;
import org.apache.flink.connector.kafka.sink.Kafkasink;
import org.apache.flink.streaming.api.datastream.DataStreamSource;
import
org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;

/**
 * @Description :
 * @School:优极限学堂
 * @Official-Website: http://www.yjxxt.com
 * @Teacher:李毅大帝
 * @Mail:863159469@qq.com
 */
public class Hello09sinkkafka {
    public static void main(String[] args) throws Exception {
        //获取程序运行的环境

```

```

        StreamExecutionEnvironment environment =
StreamExecutionEnvironment.getExecutionEnvironment();
        //写出数据
        DataStreamSource<String> source =
environment.socketTextStream("localhost", 19523);
        //设置Kafka写出配置
        KafkaSink<String> kafkasink = KafkaSink.<String>builder()
.setBootstrapServers("node01:9092,node02:9092,node03:9092")

.setRecordSerializer(KafkaRecordSerializationSchema.builder()
.setTopic("dwd_wordcount")
.setValueSerializationSchema(new
SimpleStringSchema())
.build())
)
.setDeliveryGuarantee(DeliveryGuarantee.AT_LEAST_ONCE)
.build();

//将数据写出到Kafka
source.map(word -> "yjxxt-" + word).sinkTo(kafkasink);

//开始执行
environment.execute();
}
}

```

- JDBC Sink

- 该连接器可以向 JDBC 数据库写入数据。

```

import org.apache.commons.lang3.RandomStringUtils;
import org.apache.flink.connector.jdbc.JdbcConnectionOptions;
import org.apache.flink.connector.jdbc.JdbcSink;
import org.apache.flink.streaming.api.datastream.DataStreamSource;
import
org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.functions.sink.SinkFunction;

public class Hello11sinkJDBC {
    public static void main(String[] args) throws Exception {
        //获取程序运行的环境
        StreamExecutionEnvironment environment =
StreamExecutionEnvironment.getExecutionEnvironment();

        DataStreamSource<Integer> source = environment.fromElements(11,
22, 33);
        //设置JDBC写出配置
        SinkFunction<Integer> jdbcsink = JdbcSink.sink(
                "insert into t_flink (id, num) values (?,?)",
                (ps, t) -> {
                    ps.setString(1, "yjxxt_" +
System.currentTimeMillis() + "_" +
RandomStringUtils.randomAlphanumeric(5));
                    ps.setInt(2, t);
                },
                new
JdbcConnectionOptions.JdbcConnectionOptionsBuilder()

```

```

        .withDriverName("com.mysql.cj.jdbc.Driver")
        .withUrl("jdbc:mysql://localhost:3306/yjxxt?
useSSL=false&useUnicode=true&characterEncoding=UTF8&serverTimezone=GMT"
)
        .withUsername("root")
        .withPassword("123456")
        .build()
);
source.addSink(jdbcSink);

environment.execute();
}
}

```

6.5. 自定义Sink:

- 我们也可以通过自定义的方式，来实现我们自己的sink，自定义可以通过两种方式：
 - 实现SinkFunction接口
 - 继承RichSinkFunction类
- 代码实现

```

○ import com.yjxxt.util.UTIL;
import org.apache.commons.io.FileUtils;
import org.apache.flink.streaming.api.datastream.DataStreamSource;
import
org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.functions.sink.SinkFunction;

import java.io.File;
import java.util.ArrayList;

/**
 * @Description :
 * @School:优极限学堂
 * @Official-website: http://www.yjxxt.com
 * @Teacher:李毅大帝
 * @Mail:863159469@qq.com
 */
public class Hello15sinkCustom {
    public static void main(String[] args) throws Exception {
        //运行环境
        StreamExecutionEnvironment environment =
StreamExecutionEnvironment.getExecutionEnvironment();
        environment.setParallelism(2);
        //操作数据
        ArrayList<String> list = new ArrayList<>();
        list.add("君子周而不比，小人比而不周");
        list.add("君子喻于义，小人喻于利");
        list.add("君子怀德，小人怀土；君子怀刑，小人怀惠");
        list.add("君子欲讷于言而敏于行");
        list.add("君子坦荡荡，小人长戚戚");
        DataStreamSource<String> source =
environment.fromCollection(list);

        //写出数据
        source.addSink(new YjxxtCustomSink("data/sink" +
System.currentTimeMillis())).setParallelism(1);
    }
}

```

```

        //运行环境
        environment.execute();
    }

}

class YjxxtCustomsink implements SinkFunction<String> {

    private File file;

    public YjxxtCustomsink(String filePath) {
        this.file = new File(filePath);
    }

    @Override
    public void invoke(String line, Context context) throws Exception {
        //加密数据
        String encrypt = DESUtil.encrypt("yjxxt0523", line) + "\r\n";
        //写出数据
        FileUtils.writeStringToFile(file, encrypt, "utf-8", true);
    }
}

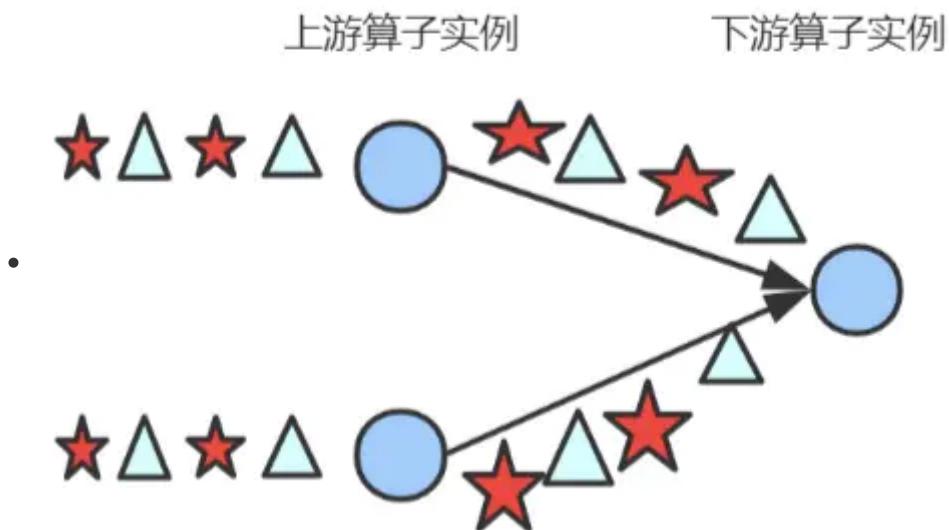
```

7. Flink Partitioner类算子

- Flink 也提供以下方法让用户根据需要在数据转换完成后对数据分区进行更细粒度的配置。
- 分区算子：用于指定上游 task 的各并行 subtask 与下游 task 的 subtask 之间如何传输数据。

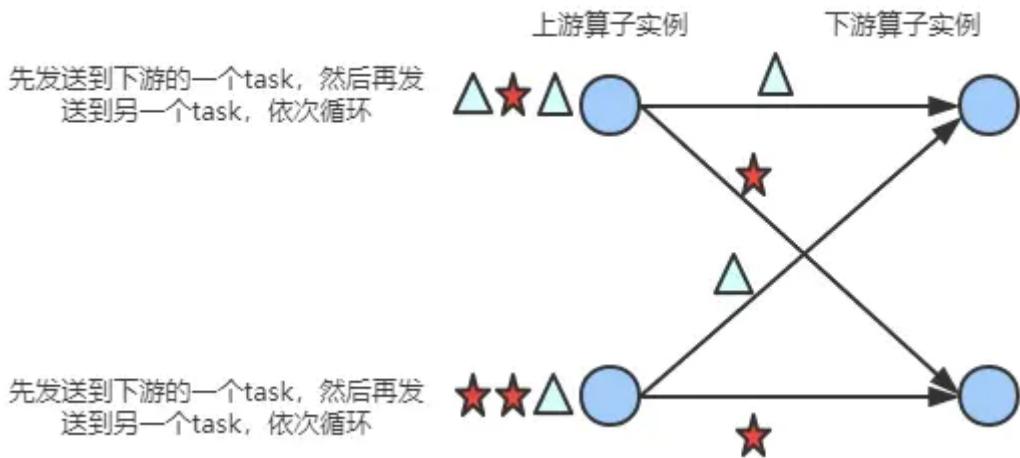
7.1. GlobalPartitioner

- 分区器会将上游所有元素都发送到下游的第一个算子实例上(SubTask Id = 0)



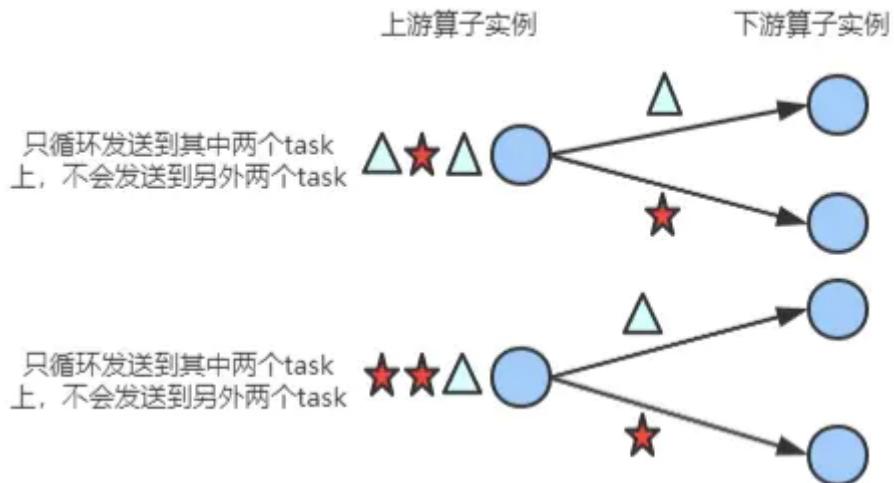
7.2. RebalancePartitioner

- 数据会被循环发送到下游的每一个实例中进行处理。



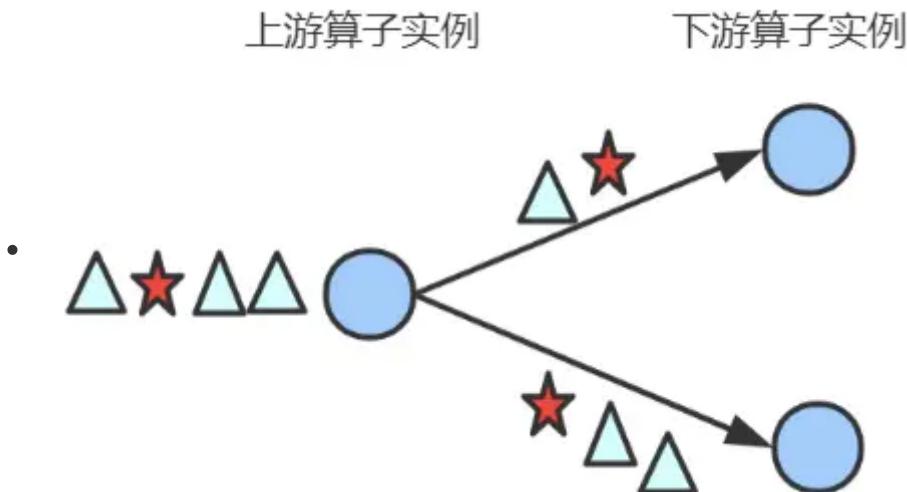
7.3. RescalePartitioner

- 这种分区器会根据上下游算子的并行度，循环的方式输出到下游算子的每个实例。
- 举例：
 - 若上游并行度是2，下游是4，则上游一个并行度以循环的方式将记录输出到下游的两个并行度上；上游另一个并行度以循环的方式将记录输出到下游另两个并行度上。
 - 若上游并行度是4，下游是2，则上游两个并行度将记录输出到下游一个并行度上；上游另两个并行度将记录输出到下游另一个并行度上
-



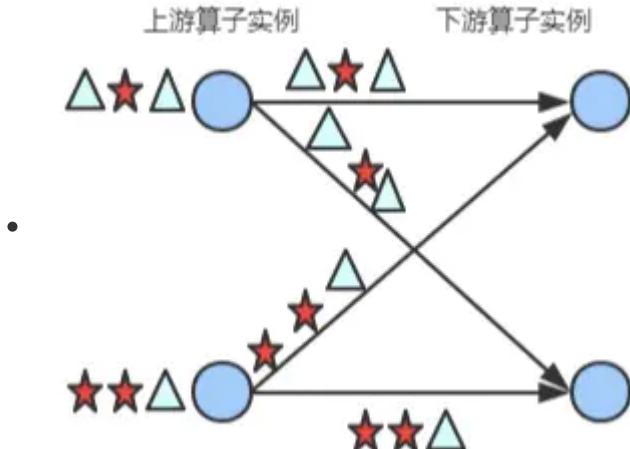
7.4. ShufflePartitioner

- 随机选择一个下游算子实例进行发送



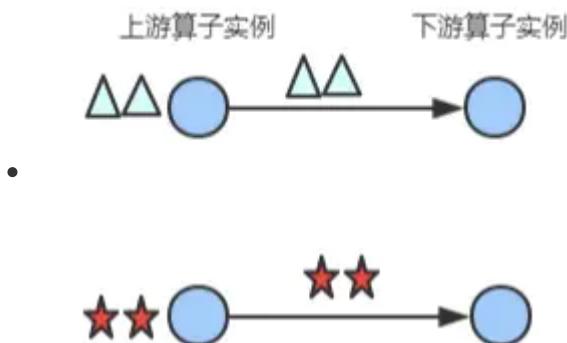
7.5. BroadcastPartitioner

- 广播分区会将上游数据输出到下游算子的每个实例中。适合于大数据集和小数据集做Join的场景。



7.6. ForwardPartitioner

- 发送到下游对应的第一个task。它要求上下游算子并行度一样。



7.7. KeyGroupStreamPartitioner

- 分区器。会将数据按Key的Hash值输出到下游算子实例中。
- Hash分区器。会将数据按 Key 的 Hash 值输出到下游算子实例中。

7.8. CustomPartitioner

- 用户自定义分区器。需要用户自己实现Partitioner接口，来定义自己的分区逻辑
- 代码实现

```
import org.apache.flink.api.common.functions.Partitioner;
import org.apache.flink.api.common.functions.RichMapFunction;
import org.apache.flink.api.java.functions.KeySelector;
import org.apache.flink.streaming.api.datastream.DataStreamSource;
import
org.apache.flink.streaming.api.datastream.SingleOutputStreamOperator;
import
org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;

/**
 * @Description :
 * @School:优极限学堂
 * @Official-Website: http://www.yjxxt.com
 * @Teacher:李毅大帝
 * @Mail:863159469@qq.com
 */
```

```

*/
public class Hello12PartitioningCommon {
    public static void main(String[] args) throws Exception {
        //获取程序运行的环境
        StreamExecutionEnvironment environment =
StreamExecutionEnvironment.getExecutionEnvironment();

        //数据源
        DataStreamSource<String> source =
environment.readTextFile("data/wordcount.txt").setParallelism(1);
        SingleOutputStreamOperator<String> upperStream = source.map(new
RichMapFunction<String, String>() {
            @Override
            public String map(String s) throws Exception {
                return "上游TaskID[" +
getRuntimeContext().getIndexOfThisSubtask() + "]" + s;
            }
        }).setParallelism(2);
        //分区器
        //
upperStream.global().print("GlobalPartitioner").setParallelism(4);
        //
upperStream.forward().print("ForwardPartitioner").setParallelism(2);
        //
upperStream.broadcast().print("BroadcastPartitioner").setParallelism(4)
;
        //
upperStream.shuffle().print("ShufflePartitioner").setParallelism(4);
        //
upperStream.rebalance().print("RebalancePartitioner").setParallelism(4)
;
        //
upperStream.rescale().print("RescalePartitioner").setParallelism(4);
        // upperStream.keyBy(s ->
s.hashCode()).print("KeyGroupStreamPartitioner").setParallelism(4);
        upperStream.partitionCustom(new Partitioner<String>() {
            @Override
            public int partition(String s, int i) {
                return Math.abs(s.hashCode()) % i;
            }
        }, new KeySelector<String, String>() {
            @Override
            public String getKey(String s) throws Exception {
                return s;
            }
        })
        .print("CustomPartitionerWrapper").setParallelism(4);
        //执行代码
        environment.execute();
    }
}

```

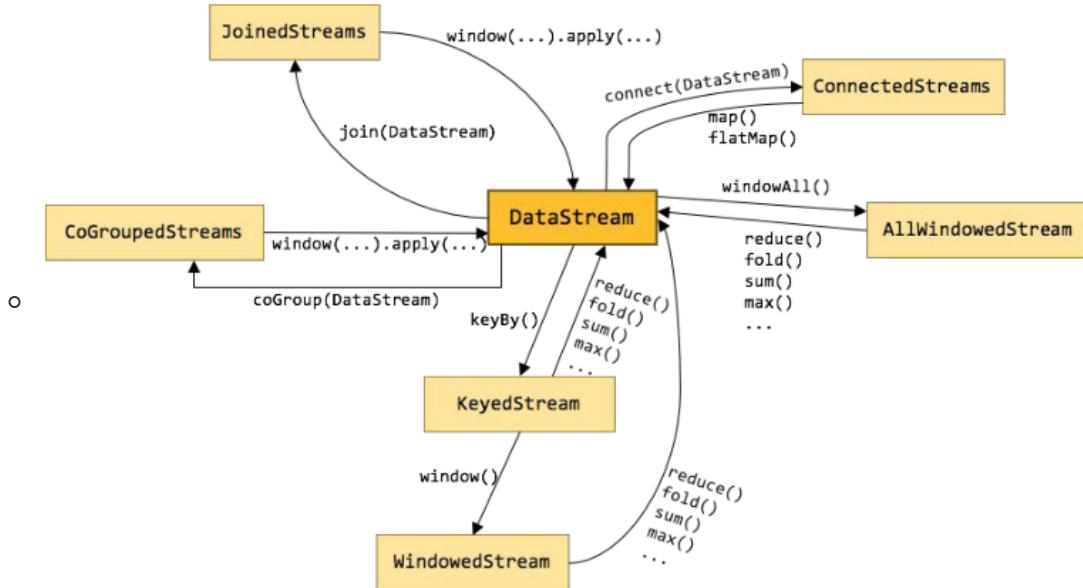
8. Flink process function

8.1. 函数介绍

- 转换算子是无法访问事件的时间戳信息和水位线信息的，而这在一些应用场景下，极为重要。
- ProcessFunction 函数是低阶流处理算子，可以访问流应用程序所有（非循环）基本构建块：
 - 事件（数据流元素）
 - 状态（容错和一致性）
 - 定时器（事件时间和处理时间）

8.2. 函数分类

- Flink 提供了 8 个不同的处理函数：



- ProcessFunction
 - 最基本的处理函数，基于 DataStream 直接调用`.process()`时作为参数传入。
- KeyedProcessFunction
 - 对流按键分区后的处理函数，基于 KeyedStream 调用`.process()`时作为参数传入。要想使用定时器，比如基于 KeyedStream。
- ProcessWindowFunction
 - 开窗之后的处理函数，也是全窗口函数的代表。基于 WindowedStream 调用`.process()`时作为参数传入。
- ProcessAllWindowFunction
 - 同样是开窗之后的处理函数，基于 AllWindowedStream 调用`.process()`时作为参数传入。
- CoProcessFunction
 - 合并（connect）两条流之后的处理函数，基于 ConnectedStreams 调用`.process()`时作为参数传入。关于流的连接合并操作
- ProcessJoinFunction
 - 间隔连接（interval join）两条流之后的处理函数，基于 IntervalJoined 调用`.process()`时作为参数传入。
- BroadcastProcessFunction
 - 广播连接流处理函数，基于 BroadcastConnectedStream 调用`.process()`时作为参数传入。
 - 这里的“广播连接流”BroadcastConnectedStream，是一个未 keyBy 的普通 DataStream 与一个广播流（BroadcastStream）做连接（connect）之后的产物。
- KeyedBroadcastProcessFunction

- 按键分区的广播连接流处理函数，同样是基于 BroadcastConnectedStream 调用 process() 时作为参数传入。
- 与 BroadcastProcessFunction 不同的是，这时的广播连接流，是一个 KeyedStream 与广播流（BroadcastStream）做连接之后的产物。
- 代码展示

```

import org.apache.flink.streaming.api.datastream.DataStreamSource;
import
org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.functions.ProcessFunction;
import org.apache.flink.util.Collector;

/**
 * @Description :
 * @School:优极限学堂
 * @Official-website: http://www.yjxxt.com
 * @Teacher:李毅大帝
 * @Mail:863159469@qq.com
 */
public class Hello13ProcessFunction {
    public static void main(String[] args) throws Exception {
        //获取程序运行的环境
        StreamExecutionEnvironment environment =
        StreamExecutionEnvironment.getExecutionEnvironment();

        //数据源
        DataStreamSource<String> source =
environment.fromElements("aa", "bb", "cc").setParallelism(1);

        //处理数据
        source.map(word -> "yjxxt_" + word).process(new
ProcessFunction<String, String>() {
            @Override
            public void processElement(String s,
ProcessFunction<String, String>.Context context, Collector<String>
collector) throws Exception {
                //查看Context
                System.out.println("[处理时间]" +
context.timerService().currentProcessingTime());
                System.out.println("[水位线/水印]" +
context.timerService().currentWatermark());
                collector.collect(s + "_" + s.hashCode());
            }
        }).print();
        //执行代码
        environment.execute();
    }
}

```

8.3. 侧输出

- process function 的 side outputs 功能可以产生多条流，并且这些流的数据类型可以不一样。
- 一个 side output 可以定义为 OutputTag[X] 对象，X 是输出流的数据类型。

- process function可以通过Context对象发射一个事件到一个或者多个side outputs。

```

• import org.apache.flink.streaming.api.datastream.DataStreamSource;
import org.apache.flink.streaming.api.datastream.SingleOutputStreamOperator;
import
org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.functions.ProcessFunction;
import org.apache.flink.util.Collector;
import org.apache.flink.util.OutputTag;

/**
 * @Description :
 * @school:优极限学堂
 * @official-website: http://www.yjxxt.com
 * @Teacher:李毅大帝
 * @Mail:863159469@qq.com
 */
public class Hello14ProcessFunctionSideOutputTag {
    public static void main(String[] args) throws Exception {
        //获取程序运行的环境
        StreamExecutionEnvironment environment =
        StreamExecutionEnvironment.getExecutionEnvironment();

        //数据源
        DataStreamSource<String> source = environment.fromElements("a", "b",
        "c", "aa", "bb", "cc", "aaa", "bbb", "ccc");

        //处理数据,分别获取长度为2和长度为3的
        // DataStream<String> str2 = source.filter(w -> w.length() == 2);
        // DataStream<String> str3 = source.filter(w -> w.length() == 3);

        //侧输出获取数据
        OutputTag<String> outputTag2 = new OutputTag<String>("sideOutput2")
    {
        };
        OutputTag<String> outputTag3 = new OutputTag<String>("sideOutput3")
    {
        };
        SingleOutputStreamOperator<String> processStream =
        source.process(new ProcessFunction<String, String>() {
            @Override
            public void processElement(String s, ProcessFunction<String,
String>.Context context, Collector<String> collector) throws Exception {
                //侧输出收集
                if (s != null && s.length() == 2) {
                    context.output(outputTag2, s);
                } else if (s != null && s.length() == 3) {
                    context.output(outputTag3, s);
                }
                //主线收集
                collector.collect(s.toUpperCase());
            }
        });
        //主线剧情
        // processStream.print().setParallelism(1);
        //获取侧输出数据
    }
}

```

```

processStream.getSideOutput(outputTag2).print("outputTag2").setParallelism(
    1);

processStream.getSideOutput(outputTag3).print("outputTag3").setParallelism(
    1);

        //执行代码
        environment.execute();
    }
}

```

9. Flink 时间语义

9.1. 宋朝大事记

宋太祖赵匡胤：

公元960年：庚申，建隆元年，辽穆宗耶律璟应历十年，后周恭帝柴宗训显德七年

公元963年：癸亥，建隆四年，乾德元年，北宋灭荆南

公元965年：乙丑，乾德三年，北宋灭后蜀

公元968年：戊辰，乾德六年，开宝元年

公元969年：己巳，开宝二年，辽穆宗耶律璟应历十九年，辽景宗耶律贤保宁元年

公元971年：辛未，开宝四年，北宋灭南汉

公元975年：乙亥，开宝八年，北宋灭南唐

公元976年：丙子，开宝九年，宋太宗赵炅改为太平兴国元年

宋太宗赵匡义：

公元976年：丙子，太平兴国元年，开宝九年

公元978年：戊寅，太平兴国三年，北宋灭吴越

公元979年：己卯，太平兴国四年，北宋灭北汉，五代十国片面完毕。

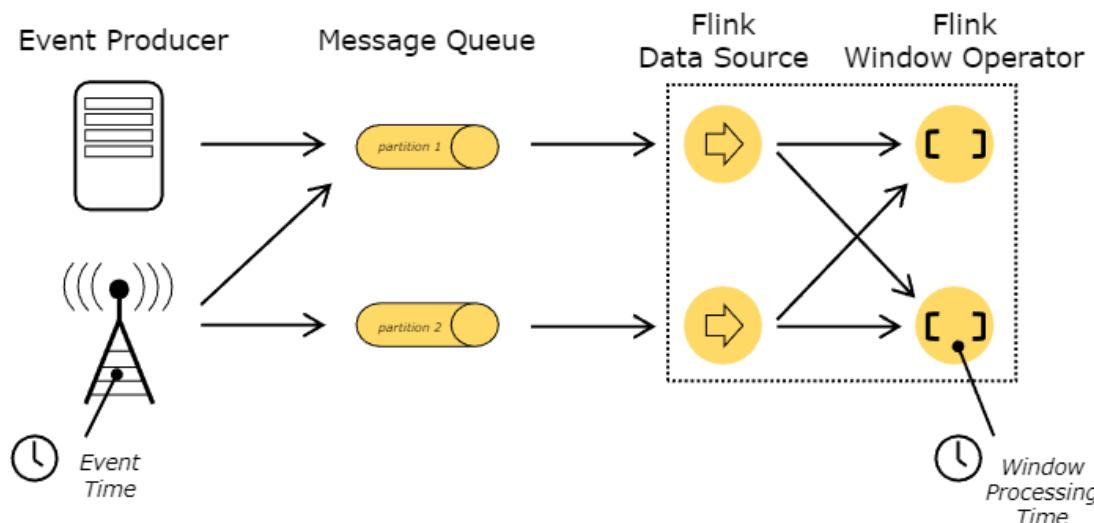
公元983年：癸未，太平兴国八年，辽景宗耶律贤乾亨五年，辽圣宗耶律隆绪统和元年

公元984年：甲申，太平兴国九年，雍熙元年

- 时间点：

- 历史事件发生的时间，永远不会改变
- 历史事件回顾的时间

9.2. 时间概念



- 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。这个时间通常在记录进入Flink之前嵌入在记录中，并且可以从每个记录中提取事件时间戳。
 - 在事件时间中，时间的进展取决于数据，而不是任何挂钟。事件时间程序必须指定如何生成事件时间水印，这是表示事件时间进度的机制。
 - 事件时间指的是数据本身携带的时间。这个时间是在事件产生时的时间。
 - 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。
 - 例如：充值数据
- Ingestion time(摄入时间)
 - 摄入时间指的是数据进入 Flink 的时间；
- Processing time (处理时间)
 - 处理时间是指正在执行相应操作的机器的系统时间。
 - 当流式程序在基于处理时间运行时，所有基于时间的操作（如时间窗口）都将使用运行相应的机器的系统时钟。
 - 每小时处理时间窗口将包括在系统时钟指示整小时的时间之间到达特定操作员的所有记录。
 - 如果应用程序在上午9:15开始运行，第一个每小时处理时间窗口将包括上午9:15和上午10:00之间处理的事件
 - 下一个窗口将包括在上午10:00和上午11:00之间处理的活动，以此类推。
 - 处理时间是最简单的时间概念，不需要流和机器之间的协调。它提供了最佳的性能和最低的延迟。但是，在分布式和异步环境中，处理时间不提供确定性，因为它容易受到记录到达系统的速度（例如，从消息队列）、记录在系统内的操作员之间流动的速度以及中断（计划的或其他）的影响。
 - 例如：秒杀数据

9.3. 代码实现

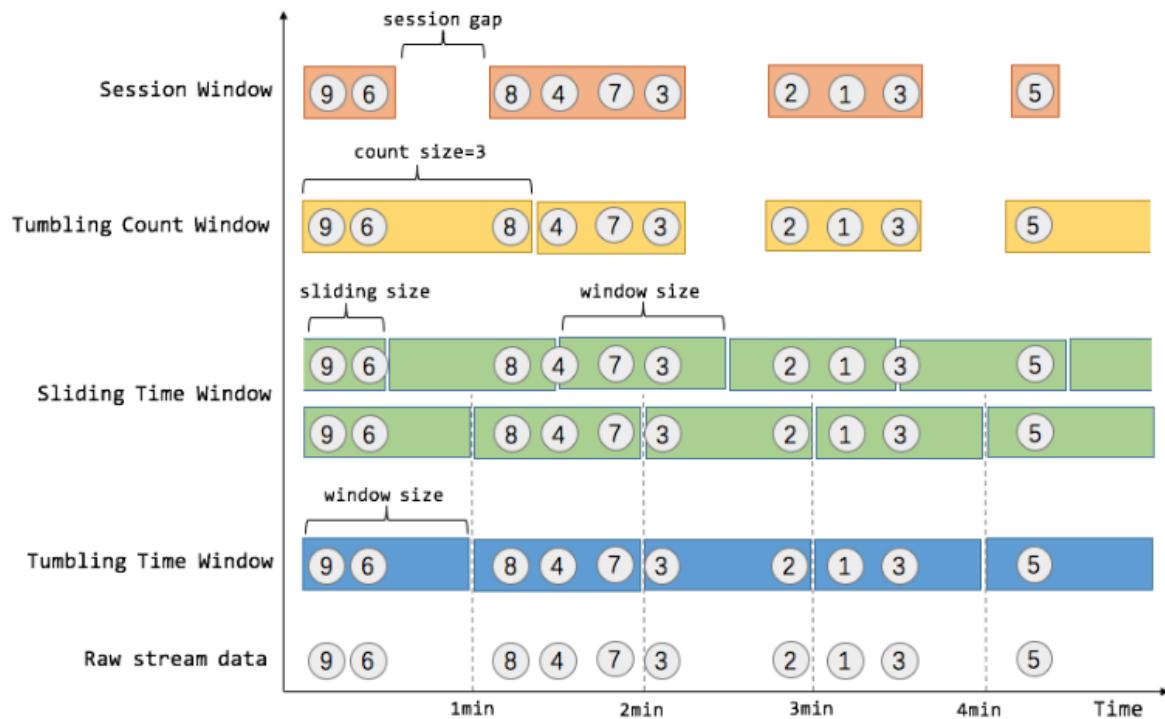
- 时间语义主要为窗口计算而服务
- 1.12 以前，flink 默认以 processing time 作为默认的时间语义；可以在 env 上设置所想要的时间语义；但是新版本已经deprecated 了上述设置 api；

- ```
//设置 EventTime 作为时间标准
environment.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);
//设置 IngestionTime 作为时间标准
environment.setStreamTimeCharacteristic(TimeCharacteristic.IngestionTime);
//设置 ProcessingTime 作为时间标准
environment.setStreamTimeCharacteristic(TimeCharacteristic.ProcessingTime);
```

- 1.12 及以后，flink 以 event time 作为默认的时间语义

- ```
keyedStream.window(SlidingEventTimeWindows.of(Time.seconds(5),
Time.seconds(1)));
keyedStream.window(SlidingProcessingTimeWindows.of(Time.seconds(5),
Time.seconds(1)));
keyedStream.window(TumblingEventTimeWindows.of(Time.seconds(5)));
keyedStream.window(TumblingProcessingTimeWindows.of(Time.seconds(5)));
keyedStream.window(EventTimeSessionWindows.withGap(Time.minutes(10)));
keyedStream.window(ProcessingTimeSessionWindows.withGap(Time.minutes(10)));
```

10. Flink 窗口函数



10.1. 基本概念

- 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。
- 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。
- 窗口分类：



- 基于时间划分驱动（Time Window）例如：每30秒钟
- 基于数据数量驱动（Count Window）例如：每一百个元素，与时间无关。

10.2. keyed&non-keyed

- Flink 窗口在 *keyed streams* 和 *non-keyed streams* 上使用的基本结构：
 - keyed streams 要调用 `keyBy(...)` 后再调用 `window(...)`，
 - non-keyed streams 只用直接调用 `windowAll(...)`。
 -

Keyed Windows

```
stream
    .keyBy(...)
        <- 仅 keyed 窗口需要
    .window(...)
        <- 必填项: "assigner"
    [.trigger(...)]      <- 可选项: "trigger" (省略则使用默认 trigger)
    [.evictor(...)]     <- 可选项: "evictor" (省略则不使用 evictor)
    [.allowedLateness(...)] <- 可选项: "lateness" (省略则为 0)
    [.sideOutputLateData(...)] <- 可选项: "output tag" (省略则不对迟到数据使用 side output)
    .reduce/aggregate/apply()
        <- 必填项: "function"
    [.getSideOutput(...)]   <- 可选项: "output tag"
```

Non-Keyed Windows

```
stream
    .windowAll(...)
        <- 必填项: "assigner"
    [.trigger(...)]      <- 可选项: "trigger" (else default trigger)
    [.evictor(...)]     <- 可选项: "evictor" (else no evictor)
    [.allowedLateness(...)] <- 可选项: "lateness" (else zero)
    [.sideOutputLateData(...)] <- 可选项: "output tag" (else no side output for late data)
    .reduce/aggregate/apply()
        <- 必填项: "function"
    [.getSideOutput(...)]   <- 可选项: "output tag"
```

- 概述：

在实际案例中Keyed Window 使用最多，所以我们需要掌握Keyed window的算子，在每个窗口算子中包含了

Windows Assigner、**Windows Trigger**(窗口触发器)、**Evictor**(数据剔除器)、**Lateness**(时延设定)、

Output (输出标签)以及**windows Function**, 其中**windows Assigner**和**windows Functions**是所有窗口算子

必须指定的属性，其余的属性都是根据实际情况选择指定。

code:

```
stream.keyBy(...)是Keyed类型数据集
>window(...)//指定窗口分配器类型
[.trigger(...)]//指定触发器类型(可选)
[.evictor(...)] // 指定evictor或者不指定(可选)
[.allowedLateness(...)] //指定是否延迟处理数据(可选)
[.sideOutputLateData(...)] // 指定output tag(可选)
.reduce/aggregate/fold/apply() //指定窗口计算函数
[.getSideOutput(...)] //根据Tag输出数据(可选)
```

intro:

Windows Assigner : 指定窗口的类型, 定义如何将数据流分配到一个或多个窗口

Windows Trigger : 指定窗口触发的时机, 定义窗口满足什么样的条件触发计算

Evictor : 用于数据剔除

allowedLateness : 标记是否处理迟到数据, 当迟到数据达到窗口是否触发计算

Output Tag: 标记输出标签, 然后在通过**getSideOutput**将窗口中的数据根据标签输出

Windows Function: 定义窗口上数据处理的逻辑, 例如对数据进行**Sum**操作

- 定义窗口前确定的是你的 stream 是 keyed 还是 non-keyed:

- keyed:

- 对于 keyed stream, 其中数据的任何属性都可以作为 key。属于同一个 key 的元素会被发送到同一个 task。
 - 使用 keyed stream 允许你的窗口计算由多个 task 并行, 因为每个逻辑上的 keyed stream 都可以被单独处理。

- non-keyed:

- 对于 non-keyed stream, 原始的 stream 不会被分割为多个逻辑上的 stream。
 - 所以所有的窗口计算会被同一个 task 完成, 也就是 parallelism 为 1。

- **windowAssigner** 负责将 stream 中的每个数据分发到一个或多个窗口中。

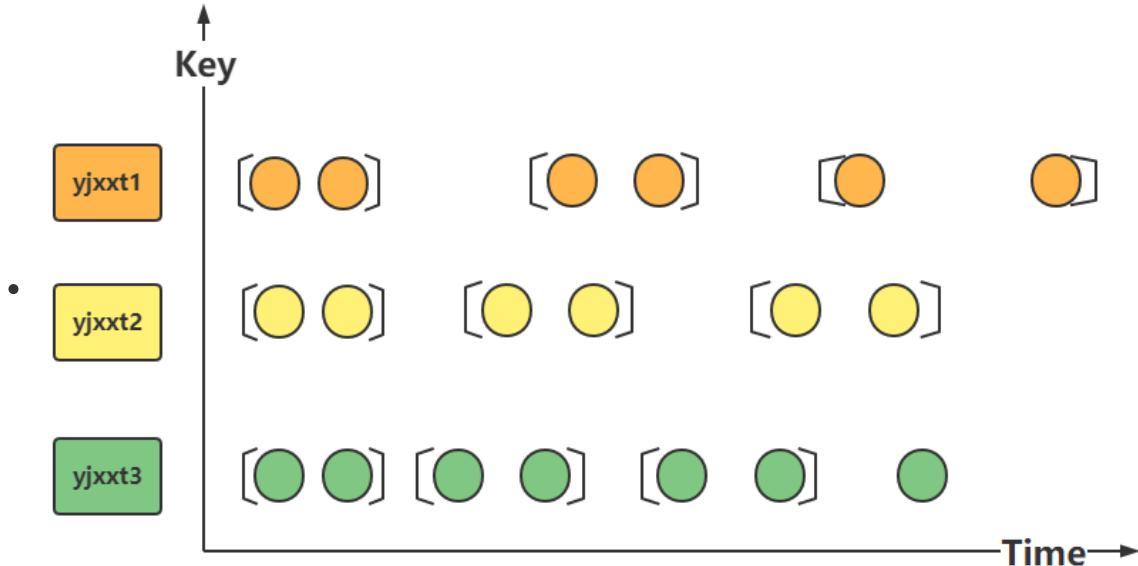
- **window(...)** (用于 keyed streams)
 - **windowAll(...)** (用于 non-keyed streams) 。

10.3. Count Window

- 计数窗口基于元素的个数来截取数据，到达固定的个数时就触发计算并关闭窗口。
- 这相当于座位有限、“人满就发车”，是否发车与时间无关。每个窗口截取数据的个数，就是窗口的大小。
- 计数窗口相比时间窗口就更加简单，我们只需指定窗口大小，就可以把数据分配到对应的窗口中了。

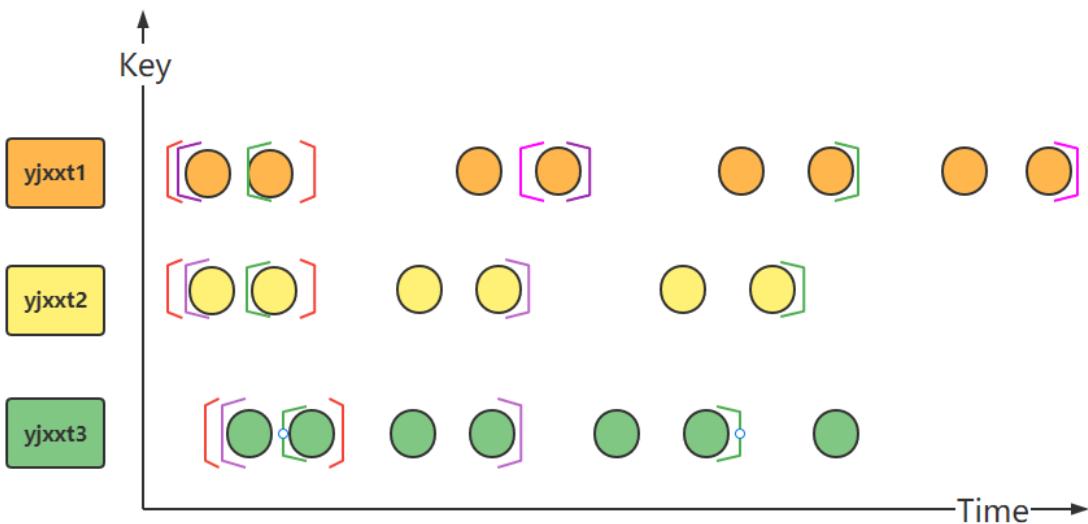
10.3.1. Tumbling Window

- 滚动窗口的 assigner 分发元素到指定大小的窗口。滚动窗口的大小是固定的，且各自范围之间不重叠。
- 滚动计数窗口只需要传入一个长整型的参数 size，表示窗口的大小。
- 使用场景：适用于按照指定的周期来统计指标。
- 比如说：每2个事件做一次统计。



10.3.2. Sliding Window

- 滑动窗口的 assigner 分发元素到指定大小的窗口，窗口大小通过 *window size* 参数设置。
- 滑动窗口需要传入两个参数：size 和 slide，前者表示窗口大小，后者表示滑动步长。
- 因此，如果 slide 小于窗口大小，滑动窗口可以允许窗口重叠。这种情况下，一个元素可能会被分发到多个窗口。
- 比如说：每5个事件做一次统计，每次滑动长度2。如下图，但图片仅供参考，与实际情况有误差
-



- 代码实现

```


- import org.apache.flink.api.common.typeinfo.Types;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.streaming.api.datastream.DataStreamSource;
import
org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;

/**
 * @Description :
 * @School:优极限学堂
 * @Official-website: http://www.yjxxt.com
 * @Teacher:李毅大帝
 * @Mail:863159469@qq.com
 */
public class Hello02Countwindow {
    public static void main(String[] args) throws Exception {
        //运行环境
        StreamExecutionEnvironment environment =
        StreamExecutionEnvironment.getExecutionEnvironment();
        //获取数据源-admin:3
        DataStreamSource<String> source =
        environment.socketTextStream("localhost", 19523);
        //Countwindow--Tumbling
        // source.map(word -> Tuple2.of(word.split(":")[0],
        Integer.parseInt(word.split(":")[1])), Types.TUPLE(Types.STRING,
        Types.INT))
        //          .keyBy(tuple2 -> tuple2.f0)
        //          .countwindow(3)
        //          .reduce((t1, t2) -> {
        //              t1.f1 = t1.f1 + t2.f1;
        //              return t1;
        //          }).print("CountWindow--"
        Tumbling:"").setParallelism(1);
        //
        //Countwindow--sliding
        source.map(word -> Tuple2.of(word.split(":")[0],
        Integer.parseInt(word.split(":")[1])), Types.TUPLE(Types.STRING,
        Types.INT))
        .keyBy(tuple2 -> tuple2.f0)
        .countwindow(3, 2)
        .reduce((t1, t2) -> {
            t1.f1 = t1.f1 + t2.f1;
        })
    }
}

```

```

        return t1;
    }).print("Countwindow--sliding:").setParallelism(1);

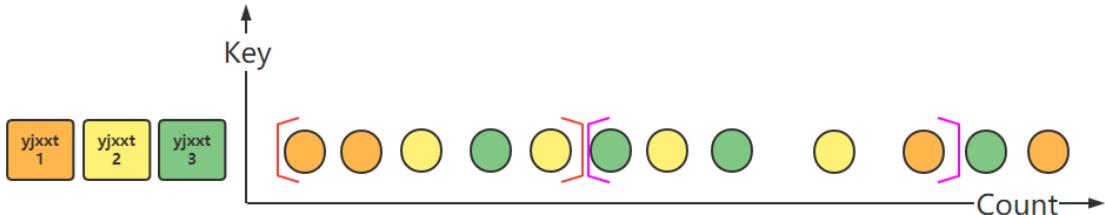
    //运行环境
    environment.execute();
}
}

```

10.3.3. Window All

- countWindowAll 数量窗口 (不分区数量滚动窗口【滑动窗口与滚动窗口的区别，在于滑动窗口会有数据元素重叠可能，而滚动窗口不存在元素重叠】)

-



- 代码实现

```

○ import org.apache.flink.api.common.typeinfo.Types;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.streaming.api.datastream.DataStreamSource;
import
org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;

/**
 * @Description :
 * @School:优极限学堂
 * @official-website: http://www.yjxxt.com
 * @Teacher:李毅大帝
 * @Mail:863159469@qq.com
 */
public class Hello03CountwindowAll {
    public static void main(String[] args) throws Exception {
        //运行环境
        StreamExecutionEnvironment environment =
        StreamExecutionEnvironment.getExecutionEnvironment();
        //获取数据源-admin:3
        DataStreamSource<String> source =
        environment.socketTextStream("localhost", 19523);
        //CountwindowAll--Tumbling
        // source.map(word -> Tuple2.of(word.split(":")[0],
        Integer.parseInt(word.split(":")[1])), Types.TUPLE(Types.STRING,
        Types.INT))
        //          .countwindowAll(3)
        //          .reduce((t1, t2) -> {
        //              t1.f0= t1.f0+"_"+t2.f0;
        //              t1.f1 = t1.f1 + t2.f1;
        //              return t1;
        //          }).print("CountwindowAll--"
        Tumbling:").setParallelism(1);

        //Countwindow--Sliding
    }
}

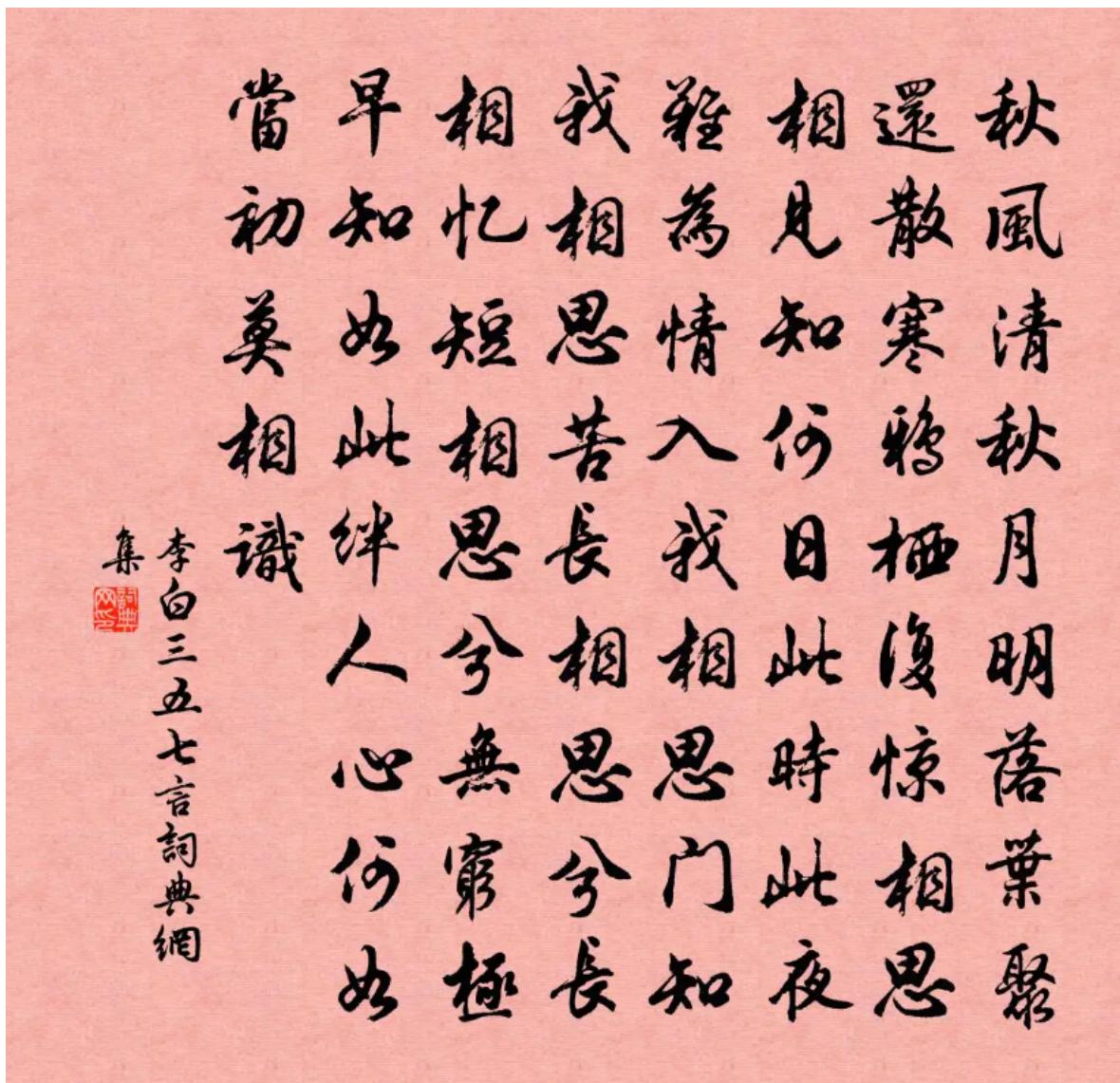
```

```

        source.map(word -> Tuple2.of(word.split(":")[0],
Integer.parseInt(word.split(":")[1])), Types.TUPLE(Types.STRING,
Types.INT))
        .countWindowAll(3, 2)
        .reduce((t1, t2) -> {
            t1.f0 = t1.f0 + "_" + t2.f0;
            t1.f1 = t1.f1 + t2.f1;
            return t1;
        }).print("CountwindowAll--sliding:").setParallelism(1);
//运行环境
environment.execute();
}
}

```

10.3.4. Minor Defects



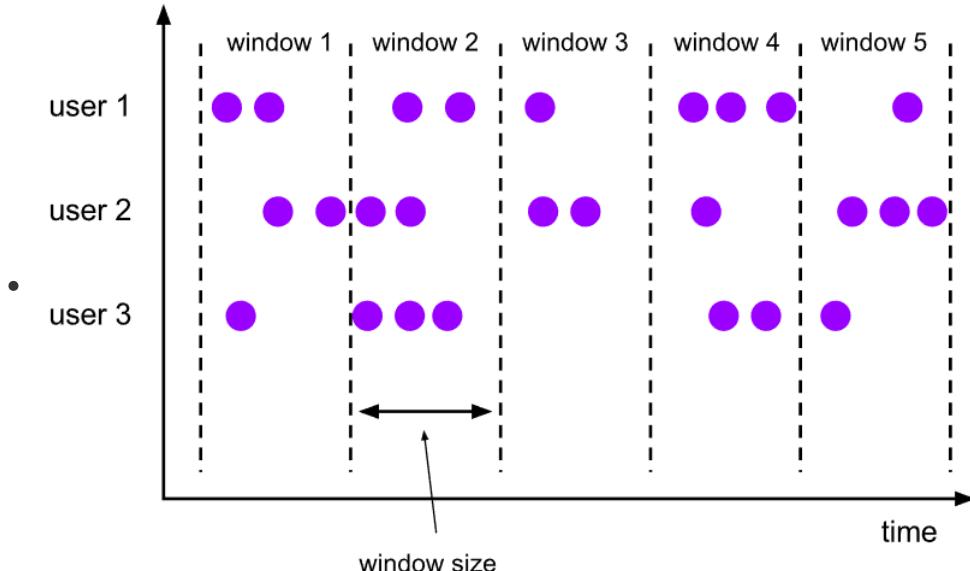
10.4. Time Window

- 时间窗口是最常用的窗口类型，又可以细分为滚动、滑动和会话三种。
 - 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)
- 除了Flink自定义的，还可以继承 `windowAssigner` 类来实现自定义的 window assigner。

- 所有内置的 window assigner (除了 global window) 都是基于时间分发数据的, processing time 或 event time 均可。

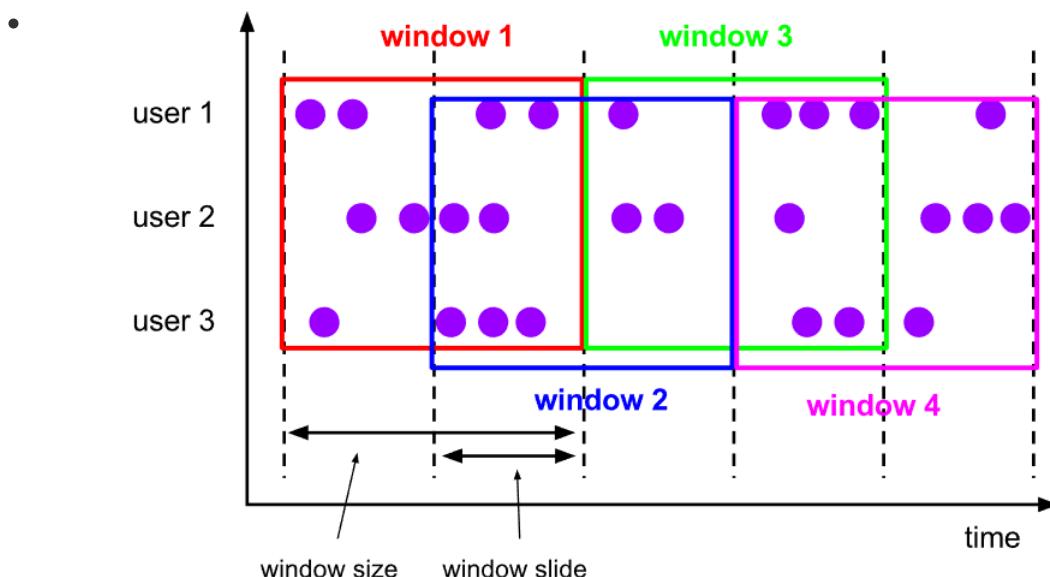
10.4.1. Tumbling Window

- 滚动窗口的 assigner 分发元素到指定大小的窗口。滚动窗口的大小是固定的, 且各自范围之间不重叠。
- 比如说, 如果你指定了滚动窗口的大小为 5 分钟, 那么每 5 分钟就会有一个窗口被计算, 且一个新的窗口被创建 (如下图所示)。



10.4.2. Sliding Window

- 滑动窗口的 assigner 分发元素到指定大小的窗口, 窗口大小通过 *window size* 参数设置。
- 滑动窗口需要一个额外的滑动距离 (*window slide*) 参数来控制生成新窗口的频率。
- 因此, 如果 *slide* 小于窗口大小, 滑动窗口可以允许窗口重叠。这种情况下, 一个元素可能会被分发到多个窗口。
- 比如说, 你设置了大小为 10 分钟, 滑动距离 5 分钟的窗口, 你会在每 5 分钟得到一个新的窗口, 里面包含之前 10 分钟到达的数据 (如下图所示)。



- 代码实现

- `import org.apache.flink.api.common.typeinfo.Types;`

```
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.streaming.api.datastream.DataStreamSource;
import
org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import
org.apache.flink.streaming.api.windowing.assigners.SlidingProcessingTim
ewindows;
import org.apache.flink.streaming.api.windowing.time.Time;

import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;

/**
 * @Description :
 * @School:优极限学堂
 * @official-website: http://www.yjxxt.com
 * @Teacher:李毅大帝
 * @Mail:863159469@qq.com
 */
public class Hello04Timewindow {
    public static void main(String[] args) throws Exception {
        //运行环境
        StreamExecutionEnvironment environment =
StreamExecutionEnvironment.getExecutionEnvironment();
        //获取数据源-admin:3
        DataStreamSource<String> source =
environment.socketTextStream("localhost", 19523);
        //Timewindow--Tumbling
        // source.map(word -> Tuple2.of(word.split(":")[0],
Integer.parseInt(word.split(":")[1])), Types.TUPLE(Types.STRING,
Types.INT))
        //           .keyBy(tuple2 -> tuple2.f0)
        //
        .window(TumblingProcessingTimeWindows.of(Time.seconds(5)))
        //           .reduce((t1, t2) -> {
        //               t1.f1 = t1.f1 + t2.f1;
        //               return t1;
        //           })
        //           .map(tuple2 -> {
        //               tuple2.f0 =
LocalDateTime.now().format(DateTimeFormatter.ofPattern("yyyy年MM月dd日HH
时mm分ss秒SSS毫秒")) + tuple2.f0;
        //               return tuple2;
        //           }, Types.TUPLE(Types.STRING, Types.INT))
        //           .print("Timewindow--Tumbling:").setParallelism(1);

        //Timewindow--Sliding
        source.map(word -> Tuple2.of(word.split(":")[0],
Integer.parseInt(word.split(":")[1])), Types.TUPLE(Types.STRING,
Types.INT))
        .keyBy(tuple2 -> tuple2.f0)

        .window(SlidingProcessingTimeWindows.of(Time.seconds(5),
Time.seconds(2)))
        .reduce((t1, t2) -> {
            t1.f1 = t1.f1 + t2.f1;
            return t1;
        })
    }
}
```

```

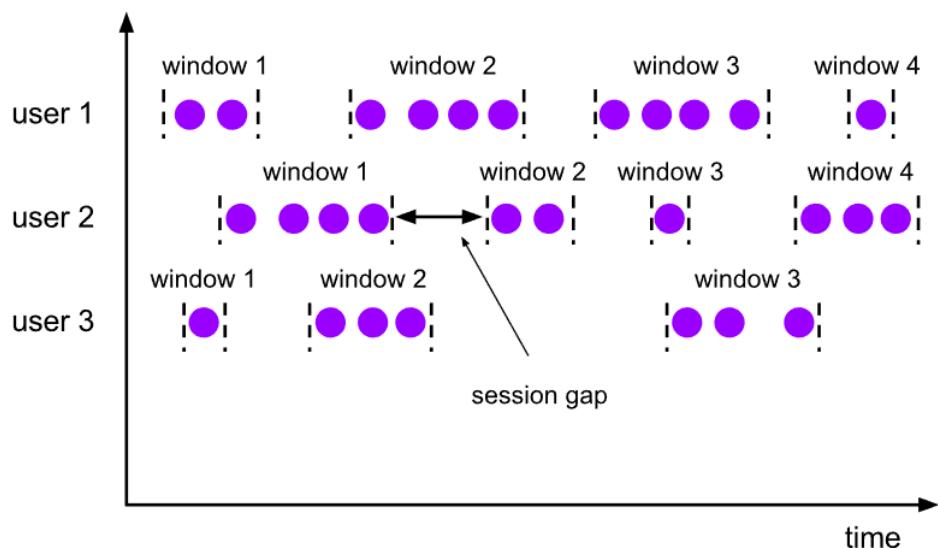
        .map(tuple2 -> {
            tuple2.f0 =
LocalDateTime.now().format(DateTimeFormatter.ofPattern("yyyy年MM月dd日HH
时mm分ss秒SSS毫秒")) + tuple2.f0;
            return tuple2;
}, Types.TUPLE(Types.STRING, Types.INT))
.print("Timewindow--Sliding:").setParallelism(1);

//运行环境
environment.execute();
}
}

```

10.4.3. Session Window

- 会话窗口的 assigner 会把数据按活跃的会话分组。
- 与滚动窗口和滑动窗口不同，会话窗口不会相互重叠，且没有固定的开始或结束时间。
- 会话窗口的 assigner 可以设置固定的会话间隔 (session gap) 或用 session gap extractor 函数来动态地定义多长时间算作不活跃。
- 当超出了不活跃的时间段，当前的会话就会关闭，并且将接下来的数据分发到新的会话窗口。
-



- 代码实现

```

○ import org.apache.flink.api.common.typeinfo.Types;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.streaming.api.datastream.DataStreamSource;
import
org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import
org.apache.flink.streaming.api.windowing.assigners.ProcessingTimeSession
windows;
import org.apache.flink.streaming.api.windowing.time.Time;

import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;

/**
 * @Description :
 * @School:优极限学堂

```

```

* @Official-website: http://www.yjxxt.com
* @Teacher: 李毅大帝
* @Mail: 863159469@qq.com
*/
public class Hello05TimewindowSession {
    public static void main(String[] args) throws Exception {
        //运行环境
        StreamExecutionEnvironment environment =
StreamExecutionEnvironment.getExecutionEnvironment();
        //获取数据源-admin:3
        DataStreamSource<String> source =
environment.socketTextStream("localhost", 19523);

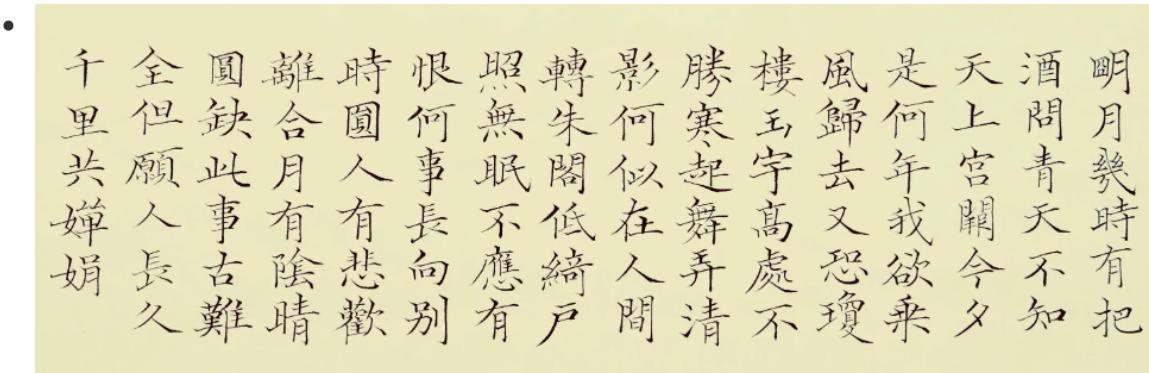
        //Timewindow--Session
        source.map(word -> Tuple2.of(word.split(":")[0],
Integer.parseInt(word.split(":")[1])), Types.TUPLE(Types.STRING,
Types.INT))
            .keyBy(tuple2 -> tuple2.f0)

        .window(ProcessingTimeSessionWindows.withGap(Time.seconds(5)))
            .reduce((t1, t2) -> {
                t1.f1 = t1.f1 + t2.f1;
                return t1;
            })
            .map(tuple2 -> {
                tuple2.f0 =
LocalDateTime.now().format(DateTimeFormatter.ofPattern("yyyy年MM月dd日HH
时mm分ss秒SSS毫秒")) + tuple2.f0;
                return tuple2;
            }, Types.TUPLE(Types.STRING, Types.INT))
            .print("Timewindow--Session:").setParallelism(1);

        //运行环境
        environment.execute();
    }
}

```

10.4.4. Minor Defects



- 代码实现

```

o import com.yjxxt.util.kafkaUtil;
import org.apache.commons.lang3.RandomStringUtils;
import org.apache.flink.api.common.eventtime.WatermarkStrategy;
import org.apache.flink.api.common.serialization.SimpleStringSchema;
import org.apache.flink.api.common.typeinfo.Types;

```

```
import org.apache.flink.api.java.tuple.Tuple3;
import org.apache.flink.connector.kafka.source.KafkaSource;
import
org.apache.flink.connector.kafka.source.enumerator.initializer.OffsetsI
nitializer;
import org.apache.flink.streaming.api.datastream.DataStreamSource;
import org.apache.flink.streaming.api.datastream.KeyedStream;
import
org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import
org.apache.flink.streaming.api.windowing.assigners.TumblingEventTimeWin
dows;
import org.apache.flink.streaming.api.windowing.time.Time;

import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;

/***
 * @Description :
 * @School:优极限学堂
 * @Official-website: http://www.yjxxt.com
 * @Teacher:李毅大帝
 * @Mail:863159469@qq.com
 */
public class Hello09EventTimewindow {
    public static void main(String[] args) throws Exception {
        //启动一个线程专门发送消息给Kafka, 这样我们才有数据消费
        new Thread(() -> {
            String uname = RandomStringUtils.randomAlphabetic(8);
            for (int i = 100; i < 200; i++) {
                String date =
 LocalDateTime.now().format(DateTimeFormatter.ofPattern("yyyy年MM月dd日HH
时mm分ss秒SSS"));
                KafkaUtil.sendMsg("yjxxt", uname + i % 2 + ":" + i +
                ":" + date);
                try {
                    Thread.sleep(495);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }).start();
        //获取环境
        StreamExecutionEnvironment environment =
        StreamExecutionEnvironment.getExecutionEnvironment();

        //设置Kafka连接
        KafkaSource<String> source = KafkaSource.<String>builder()

.setBootstrapServers("node01:9092,node02:9092,node03:9092")
.setTopics("yjxxt")
.setGroupId("flink_kafkaConnector")
.setStartingOffsets(OffsetsInitializer.latest())
.setValueOnlyDeserializer(new SimpleStringSchema())
.build();
        //读取数据源
    }
}
```

```

        DataStreamSource<String> kafkaSource =
environment.fromSource(source, WatermarkStrategy.noWatermarks(),
"KafkaSource");
        KeyedStream<Tuple3<String, String, String>, String> keyedStream
= kafkaSource.map(word -> {
    String[] split = word.split(":");
    return Tuple3.of(split[0], split[1], split[2]);
}, Types.TUPLE(Types.STRING, Types.STRING,
Types.STRING))
.keyBy(t -> t.f0);

        //Timewindow--Tumbling

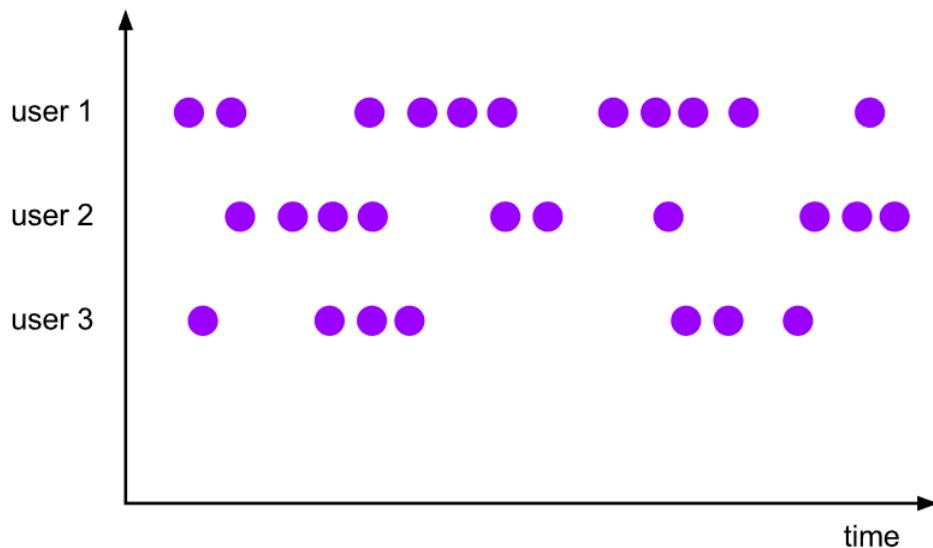
keyedStream.window(TumblingEventTimeWindows.of(Time.seconds(5)))
.reduce((t1, t2) -> {
    t1.f1 = t1.f1 + "-" + t2.f1;
    return t1;
}).map(t -> {
    t.f2 =
LocalDateTime.now().format(DateTimeFormatter.ofPattern("yyyy年MM月dd日HH
时mm分ss秒SSS"));
    return t;
}, Types.TUPLE(Types.STRING, Types.STRING,
Types.STRING))
.print("[Timewindow--Tumbling--
EventTime]").setParallelism(1);

        //执行环境
environment.execute();
}
}

```

10.5. Global Windows

- GlobalWindows作为一个全局的窗口分配器，它不像TimeWindow或CountWindow那样通过元素个数来划分成一个个窗口，而是把分区内的所有元素分配到同一个窗口，所以说如果没有定义触发器，那么整个subTask中就只有一个窗口，且一直存在，不会触发计算。
- 窗口模式仅在你指定了自定义的【trigger】时有用。否则，计算不会发生，因为全局窗口没有天然的终点去触发其中积累的数据。
- 使用Global Windows需要非常慎重，用户需要非常明确自己在整个窗口中统计出的结果是什么，并指定对应的触发器，同时还需要有指定相应的数据清理机制，否则数据将一直留在内存中。
- window和windowAll都是对stream定义窗口的方法，都需要传入WindowAssigner（窗口分配器）执行具体的开窗操作
 - window只能在已经分区的 KeyedStream 上定义，通过KeyedStream转化为 WindowedStream执行具体的开窗操作。
 - windowAll只能在未分区的DataStream上定义，调用windowAll方法后，会把DataStream转化为AllWindowedStream，并得到全局统计结果。也就是说WindowAll并行度只能1，且不可设置并行度。
-



- ```
• import org.apache.flink.api.common.typeinfo.Types;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.streaming.api.datastream.DataStreamSource;
import
org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.windowing.assigners.GlobalWindows;
import org.apache.flink.streaming.api.windowing.triggers.CountTrigger;
import org.apache.flink.streaming.api.windowing.triggers.PurgingTrigger;

/**
 * @Description :
 * @School:优极限学堂
 * @Official-Website: http://www.yjxxt.com
 * @Teacher:李毅大帝
 * @Mail:863159469@qq.com
 */
public class Hello08GlobalWindow {
 public static void main(String[] args) throws Exception {
 //运行环境
 StreamExecutionEnvironment environment =
StreamExecutionEnvironment.getExecutionEnvironment();
 //获取数据源-admin:3
 DataStreamSource<String> source =
environment.socketTextStream("localhost", 19523);
 //GlobalWindow
 source.map(word -> Tuple2.of(word.split(":")[0],
Integer.parseInt(word.split(":")[1])), Types.TUPLE(Types.STRING, Types.INT))
 .keyBy(tuple2 -> tuple2.f0)
 .window(GlobalWindows.create())
 .trigger(PurgingTrigger.of(CountTrigger.of(5)))
 .reduce((t1, t2) -> {
 t1.f1 = t1.f1 + t2.f1;
 return t1;
 }).print("GlobalWindow:").setParallelism(1);

 source.map(word -> Tuple2.of(word.split(":")[0],
Integer.parseInt(word.split(":")[1])), Types.TUPLE(Types.STRING, Types.INT))
 .windowAll(GlobalWindows.create())
 .trigger(PurgingTrigger.of(CountTrigger.of(5)))
 .reduce((t1, t2) -> {
 t1.f1 = t1.f1 + t2.f1;
 return t1;
 }).print("GlobalWindow:").setParallelism(1);
 }
}
```

```

 return t1;
 }).print("GlobalWindow:").setParallelism(1);

 //运行环境
 environment.execute();
}
}

```

## 11. Flink Window Functions

- 定义了窗口分配器，我们知道了数据属于哪个窗口，可以将数据收集起来了；至于收集起来到底要做什么，其实还完全没有头绪。
- 所以在窗口分配器之后，必须再接上一个定义窗口如何进行计算的操作，这就是所谓的“窗口函数”（window functions）
- Flink提供了两大类窗口函数，分别为增量聚合函数和全量窗口函数。
  - 增量聚合函数(incremental aggregation functions)
    - 窗口将数据收集起来，最基本的处理操作当然就是进行聚合。窗口对无限流的切分，可以看作得到了一个有界数据集。如果我们等到所有数据都收集齐，在窗口到了结束时间要输出结果的一瞬间再去进行聚合，显然就不够高效
    - 为了提高实时性，我们可以再次将流处理的思路发扬光大：就像 DataStream 的简单聚合一样，每来一条数据就立即进行计算，中间只要保持一个简单的聚合状态就可以了；区别只是在于不立即输出结果，而是要等到窗口结束时间。等到窗口到了结束时间需要输出计算结果的时候，我们只需要拿出之前聚合的状态直接输出，这无疑就大大提高了程序运行的效率和实时性。
    - 典型的增量聚合函数有ReduceFunction、AggregateFunction。
  - 全窗口聚合函数(full window functions)
    - 典型的批处理思路了--养肥了再杀
    - 全量窗口函数需要对所有进入该窗口的数据进行缓存，等到窗口触发时才会遍历窗口内所有数据，进行结果计算。
    - 因为有些场景下，我们要做的计算必须基于全部的数据才有效，这时做增量聚合就没什么意义了；另外，输出的结果有可能要包含上下文中的一些信息（比如窗口的起始时间），这是增量聚合函数做不到的。所以，我们还需要有更丰富的窗口计算方式，这就可以用全窗口函数来实现。
    - 全窗口函数也有两种：WindowFunction 和 ProcessWindowFunction。

### 11.1. 增量聚合函数

#### 11.1.1. ReduceFunction

- 最基本的聚合方式就是归约（reduce）
- 窗口函数中也提供了 ReduceFunction：只要基于 WindowedStream 调用.reduce()方法，然后传入 ReduceFunction 作为参数，就可以指定以归约两个元素的方式去对窗口中数据进行聚合了。
- ReduceFunction 可以解决大多数归约聚合的问题，但是这个接口有一个限制，就是聚合状态的类型、输出结果的类型都必须和输入数据类型一样。
- 代码实现
  - ```
import org.apache.flink.api.common.typeinfo.Types;
import org.apache.flink.api.java.tuple.Tuple2;
```

```

import org.apache.flink.streaming.api.datastream.DataStreamSource;
import
org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;

/**
 * @Description :
 * @School:优极限学堂
 * @Official-website: http://www.yjxxt.com
 * @Teacher:李毅大帝
 * @Mail:863159469@qq.com
 */
public class Hello09WindowFunctionByReduce {
    public static void main(String[] args) throws Exception {
        //运行环境
        StreamExecutionEnvironment environment =
        StreamExecutionEnvironment.getExecutionEnvironment();
        //获取数据源-admin:3
        DataStreamSource<String> source =
        environment.socketTextStream("localhost", 19523);
        //CountWindow--Tumbling--增量计算
        source.map(word -> Tuple2.of(word.split(":")[0],
        Integer.parseInt(word.split(":")[1])), Types.TUPLE(Types.STRING,
        Types.INT))
            .keyBy(tuple2 -> tuple2.f0)
            .countwindow(3)
            .reduce((t1, t2) -> {
                System.out.println("窗口增量计算函数-来一条算一条.main["
                + t1 + "][" + t2 + "]");
                t1.f0 = t1.f0 + "_" + t2.f0;
                t1.f1 = t1.f1 + t2.f1;
                return t1;
            }).print("Countwindow--Tumbling:").setParallelism(1);

        //运行环境
        environment.execute();
    }
}

```

11.1.2. AggregateFunction

- ReduceFunction 可以解决大多数归约聚合的问题，但是这个接口有一个限制，就是聚合状态的类型、输出结果的类型都必须和输入数据类型一样。这就迫使我们必须在聚合前，先将数据转换 (map) 成预期结果类型；而在有些情况下，还需要对状态进行进一步处理才能得到输出结果，这时它们的类型可能不同，使用 ReduceFunction 就会非常麻烦。
- 例如，如果我们希望计算一组数据的平均值，应该怎样做聚合呢？很明显，这时我们需要计算两个状态量：数据的总和 (sum)，以及数据的个数 (count)，而最终输出结果是两者的商 (sum/count)。如果用 ReduceFunction，那么我们应该先把数据转换成二元组(sum, count)的形式，然后进行归约聚合，最后再将元组的两个元素相除转换得到最后的平均值。本来应该只是一个任务，可我们却需要 map-reduce-map 三步操作，这显然不够高效。
- AggregateFunction 可以看作是 ReduceFunction 的通用版本，这里有三种类型：
 - 输入类型 (IN)、累加器类型 (ACC) 和输出类型 (OUT) 。
 - 输入类型 IN 就是输入流中元素的数据类型；

- 累加器类型 ACC 则是我们进行聚合的中间状态类型;
- 而输出类型当然就是最终计算结果的类型了。
- 接口中有四个方法:
 - createAccumulator():
 - 创建一个累加器，这就是为聚合创建了一个初始状态，每个聚合任务只会调用一次。
 - add():
 - 将输入的元素添加到累加器中。这就是基于聚合状态，对新来的数据进行进一步聚合的过程。方法传入两个参数:
 - 当前新到的数据 value，和当前的累加器 accumulator;
 - 返回一个新的累加器值，也就是对聚合状态进行更新。每条数据到来之后都会调用这个方法。
 - getResult():
 - 从累加器中提取聚合的输出结果。也就是说，我们可以定义多个状态，然后再基于这些聚合的状态计算出一个结果进行输出。
 - 比如之前我们提到的计算平均值，就可以把 sum 和 count 作为状态放入累加器，而在调用这个方法时相除得到最终结果。
 - 这个方法只在窗口要输出结果时调用。
 - merge():
 - 合并两个累加器，并将合并后的状态作为一个累加器返回。这个方法只在需要合并窗口的场景下才会被调用;
 - 最常见的合并窗口（Merging Window）的场景就是会话窗口（Session Windows）。

- 代码实现

```

◦ import org.apache.flink.api.common.functions.AggregateFunction;
◦ import org.apache.flink.api.common.typeinfo.Types;
◦ import org.apache.flink.api.java.tuple.Tuple2;
◦ import org.apache.flink.streaming.api.datastream.DataStreamSource;
◦ import
◦ org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;

/**
 * @Description :
 * @School:优极限学堂
 * @Official-website: http://www.yjxxt.com
 * @Teacher:李毅大帝
 * @Mail:863159469@qq.com
 */
public class Hello10WindowFunctionsByAggregate {
    public static void main(String[] args) throws Exception {
        //运行环境
        StreamExecutionEnvironment environment =
        StreamExecutionEnvironment.getExecutionEnvironment();
        //获取数据源-admin:3
        DataStreamSource<String> source =
        environment.socketTextStream("localhost", 19523);
        //Countwindow--Tumbling--增量计算
        source.map(word -> Tuple2.of(word.split(":")[0],
        Integer.parseInt(word.split(":")[1])), Types.TUPLE(Types.STRING,
        Types.INT))
            .keyBy(tuple2 -> tuple2.f0)
            .countWindow(3)
            .aggregate(new AggregateFunction<Tuple2<String,
        Integer>, Tuple2<Integer, Integer>, Double>() {
                @Override

```

```

        public Tuple2<Integer, Integer> createAccumulator()
    {
        //初始化累加器
        return Tuple2.of(0, 0);
    }

    @Override
    public Tuple2<Integer, Integer> add(Tuple2<String,
    Integer> in, Tuple2<Integer, Integer> acc) {
        acc.f0 = acc.f0 + in.f1;
        acc.f1 += 1;
        return acc;
    }

    @Override
    public Double getResult(Tuple2<Integer, Integer>
    acc) {
        //判断除数不能为0
        if (acc.f1 == 0) {
            return 0.0;
        }
        return acc.f0 * 1.0 / acc.f1;
    }

    @Override
    public Tuple2<Integer, Integer>
merge(Tuple2<Integer, Integer> integerIntegerTuple2, Tuple2<Integer,
    Integer> acc1) {
        return null;
    }
}
.print("Countwindow--Tumbling:").setParallelism(1);

//运行环境
environment.execute();
}
}

```

11.2. 全量窗口函数

11.2.1. ProcessWindowFunction

- ProcessWindowFunction 是 Window API 中最底层的通用窗口函数接口。之所以说它“最底层”，是因为除了可以拿到窗口中的所有数据之外，ProcessWindowFunction 还可以获取到一个“上下文对象”（Context）。
- 上下文对象非常强大，不仅能够获取窗口信息，还可以访问当前的时间和状态信息。这里的时间就包括了处理时间（processing time）和事件时间水位线（eventtime watermark）。
- 全量窗口的好处是以牺牲性能和资源为代价的。作为一个全窗口函数，ProcessWindowFunction 同样需要将所有数据缓存下来、等到窗口触发计算时才使用。它其实就是一个增强版的 WindowFunction。
- 代码实现

```
o import org.apache.flink.api.common.typeinfo.Types;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.api.java.tuple.Tuple3;
import org.apache.flink.streaming.api.datastream.DataStreamSource;
import
org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import
org.apache.flink.streaming.api.functions.windowing.WindowFunction;
import
org.apache.flink.streaming.api.windowing.assigners.SlidingProcessingTim
ewindows;
import org.apache.flink.streaming.api.windowing.time.Time;
import org.apache.flink.streaming.api.windowing.windows.TimeWindow;
import org.apache.flink.util.Collector;

/**
 * @Description :
 * @School:优极限学堂
 * @Official-website: http://www.yjxxt.com
 * @Teacher:李毅大帝
 * @Mail:863159469@qq.com
 */
public class Hello12WindowFunctionsByWindow {
    public static void main(String[] args) throws Exception {
        //运行环境
        StreamExecutionEnvironment environment =
        StreamExecutionEnvironment.getExecutionEnvironment();
        //获取数据源-admin:3
        DataStreamSource<String> source =
        environment.socketTextStream("localhost", 19523);

        //Timewindow--Sliding
        source.map(word -> Tuple2.of(word.split(":")[0],
        Integer.parseInt(word.split(":")[1])), Types.TUPLE(Types.STRING,
        Types.INT))
        .keyBy(tuple2 -> tuple2.f0)

        .window(SlidingProcessingTimeWindows.of(Time.seconds(5),
        Time.seconds(5)))
        .apply(new WindowFunction<Tuple2<String, Integer>,
        Tuple3<String, Integer, String>, String, TimeWindow>() {
            @Override
            public void apply(String s, TimeWindow window,
            Iterable<Tuple2<String, Integer>> input, Collector<Tuple3<String,
            Integer, String>> out) throws Exception {
                //计算总和
                int sum = 0;
                for (Tuple2<String, Integer> tuple2 : input) {
                    sum += tuple2.f1;
                }
                out.collect(Tuple3.of(s, sum,
                window.toString()));
            }
        })
        .print("Timewindow--Sliding:").setParallelism(1);

        //运行环境
        environment.execute();
    }
}
```

```
    }  
}
```

12. Flink WaterMark

WaterMark

英 [ˈwɔ:təmɑ:k] ⏺ ⓘ 美 [wɔ:tərma:rk] ⏺ ⓘ

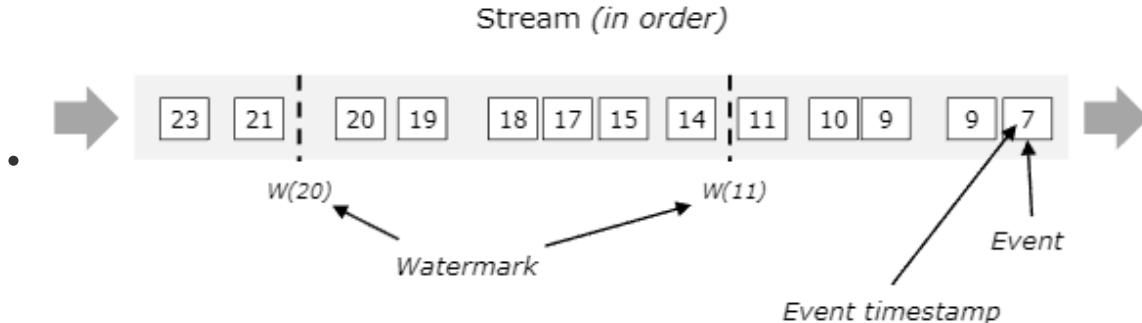
浮水印；水标志

12.1. 水位线意义

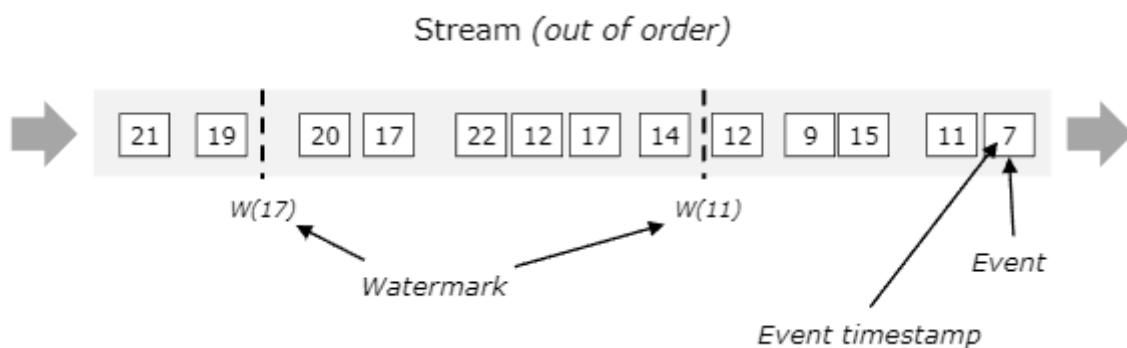
- “水位线”就是用来度量事件时间
- 如何选择时间窗口
 - 想要统计一段时间内的数据，需要划分时间窗口，这时只要判断一下事件时间就可以知道数据属于哪个窗口了。明确了一个数据的所属窗口，还不能直接进行计算。因为窗口处理的是有界数据，我们需要等窗口的数据都到齐了，才能计算出最终的统计结果。那什么时候数据就都到齐了呢？对于时间窗口来说这很明显：到了窗口的结束时间，自然就应该收集到了所有数据，就可以触发计算输出结果了。比如我们想统计 8 点~9 点的用户点击量，那就是从 8 点开始收集数据，到 9 点截止，将收集的数据做处理计算。这有点类似于班车，每小时发一班，那么 8 点之后来的人都会上同一班车，到 9 点钟准时发车；9 点之后来的人，就只好等下一班 10 点发的车了。
- 如何确认9点钟?
 - 在处理时间语义下，都是以当前任务所在节点的系统时间为准则的。这就相当于每辆车里都挂了一个钟，司机看到到了 9 点就直接发车。这种方式简单粗暴容易实现，但因为车上的钟是独立运行的，以它为标准就不能准确地判断商品的生产时间。在分布式环境下，这样会因为网络传输延迟的不确定而导致误差。比如有些商品在 8 点 59 分 59 秒生产出来，可是从下生产线到运至车上又要花费几秒，那就赶不上 9 点钟这班车了。而且现在分布式系统中有很多辆 9 点发的班车，所以同时生产出的一批商品，需要平均分配到不同班车上，可这些班车距离有近有远、上面挂的钟有快有慢，这就可能导致有些商品上车了、有些却被漏掉；先后生产出的商品，到达车上的顺序也可能乱掉：统计结果的正确性受到了影响。
- 如何给车装个表?
 - 在处理时间语义下，都是以当前任务所在节点的系统时间为准则的。这就相当于每辆车里都挂了一个钟，司机看到到了 9 点就直接发车。这种方式简单粗暴容易实现，但因为车上的钟是独立运行的，以它为标准就不能准确地判断商品的生产时间。在分布式环境下，这样会因为网络传输延迟的不确定而导致误差。比如有些商品在 8 点 59 分 59 秒生产出来，可是从下生产线到运至车上又要花费几秒，那就赶不上 9 点钟这班车了。而且现在分布式系统中有很多辆 9 点发的班车，所以同时生产出的一批商品，需要平均分配到不同班车上，可这些班车距离有近有远、上面挂的钟有快有慢，这就可能导致有些商品上车了、有些却被漏掉；先后生产出的商品，到达车上的顺序也可能乱掉：统计结果的正确性受到了影响。
- 水位线是数据流中的一部分，随着数据一起流动，在不同任务之间传输。
- 水位线可以看做是插入到数据流中的一个标记点，主要内容就是一个时间戳，用来指示当前的事件时间(实际就是用来度量事件时间的)。
- 而它插入流中的位置，就应该是在某个数据到来之后；这样就可以从这个数据中提取时间戳，作为当前水位线的时间戳了。

12.1.1. 有序流中的水位线

- 在理想状态下，数据应该按照它们生成的先后顺序、排好队进入流中；也就是说，它们处理的过程会保持原先的顺序不变，遵守先来后到的原则。这样的话我们从每个数据中提取时间戳，就可以保证总是从小到大增长的，从而插入的水位线也会不断增长、事件时钟不断向前推进。
- 实际应用中，如果当前数据量非常大，可能会有很多数据的时间戳是相同的，这时每来一条数据就提取时间戳、插入水位线就做了大量的无用功。而且即使时间戳不同，同时涌来的数据时间差会非常小（比如几毫秒），往往对处理计算也没什么影响。所以为了提高效率，一般会每隔一段时间生成一个水位线，这个水位线的时间戳，就是当前最新数据的时间戳。



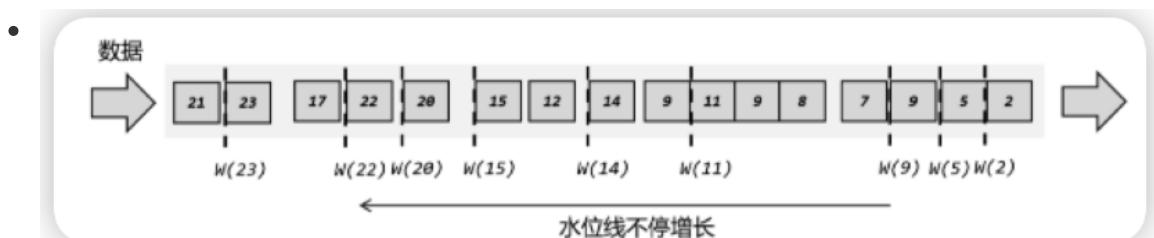
12.1.2. 乱序流中的水位线



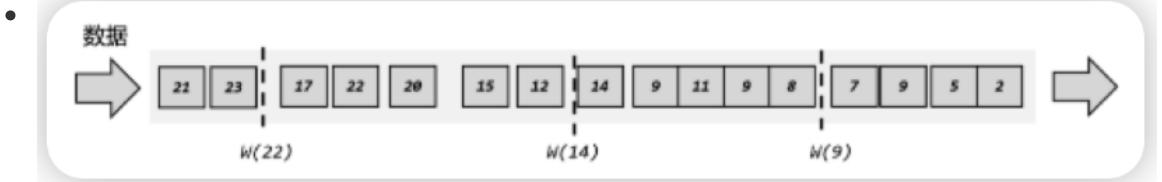
- 有序流的处理非常简单，看起来水位线也并没有起到太大的作用。但这种情况只存在于理想状态下。我们知道在分布式系统中，数据在节点间传输，会因为网络传输延迟的不确定性，导致顺序发生改变，这就是所谓的“乱序数据”。这里所说的“乱序” (out-of-order)，是指数据的先后顺序不一致，主要就是基于数据的产生时间而言的。如下图所示，一个7秒时产生的数据，生成时间自然要比9秒的数据早；但是经过数据缓存和传输之后，处理任务可能先收到了9秒的数据，之后7秒的数据才姗姗来迟。这时如果我们希望插入水位线，来指示当前的事件时间进展，又该怎么做？



- 最直观的想法自然是跟之前一样，我们还是靠数据来驱动，每来一个数据就提取它的时间戳、插入一个水位线。不过现在的情况是数据乱序，所以有可能新的时间戳比之前的还小，如果直接将这个时间的水位线再插入，我们的“时钟”就回退了——水位线就代表了时钟，时光不能倒流，所以水位线的时间戳也不能减小。解决思路也很简单：我们插入新的水位线时，要先判断一下时间戳是否比之前的大，否则就不再生成新的水位线，也就是说，只有数据的时间戳比当前时钟大，才能推动时钟前进，这时才插入水位线。

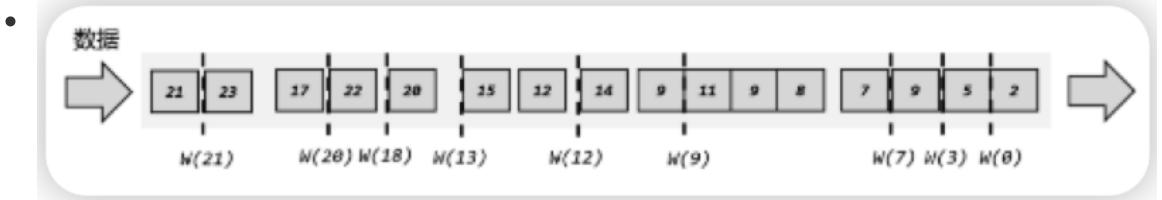


- 如果考虑到大量数据同时到来的处理效率，我们同样可以周期性地生成水位线。这时只需要保存一下之前所有数据中的最大时间戳，需要插入水位线时，就直接以它作为时间戳生成新的水位线



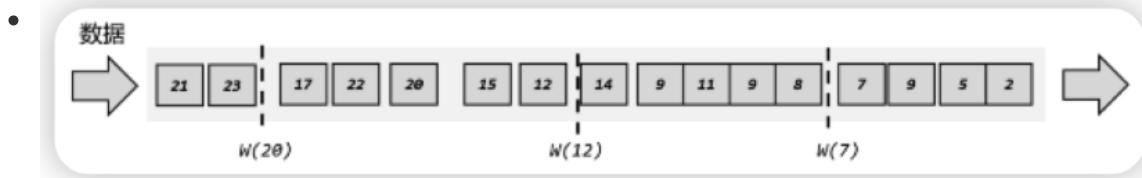
这样做尽管可以定义出一个事件时钟，却也会带来一个非常大的问题：我们无法正确处理“迟到”的数据。在上面的例子中，当9秒产生的数据到来之后，我们就直接将时钟推进到了9秒；如果有一个窗口结束时间就是9秒（比如，要统计0~9秒的所有数据），那么这时窗口就应该关闭、将收集到的所有数据计算输出结果了。但事实上，由于数据是乱序的，还可能有时间戳为7秒、8秒的数据在9秒的数据之后才到来，这就是“迟到数据”（late data）。它们本来也应该属于0~9秒这个窗口，但此时窗口已经关闭，于是这些数据就被遗漏了，这会导致统计结果不正确。如果用之前我们类比班车的例子，现在的状况就是商品不是按照生产时间顺序到来的，所以有可能出现这种情况：9点生产的商品已经到了，我们认为已经到了9点，所以直接发车；但是可能还会有8点59分59秒生产的商品迟到了，没有赶上这班车。那怎么解决这个问题呢？

其实我们有很多生活中的经验。假如是一个团队出去团建，那肯定希望每个人都不能落下；如果有人因为堵车没能准时到车上，我们可以稍微等一会儿。9点发车，我们可以等到9点10分，等人都到齐了再出发。当然，实际应用的网络环境不可能跟北京的交通一样堵，所以不需要等那么久，或许只要等一两秒钟就可以了。具体在商品班车的例子里，我们可以多等2秒钟，也就是当生产时间为9点零2秒的商品到达，时钟推进到9点零2秒，这时就认为所有8点到9点生产的商品都到齐了，可以正式发车。不过这样相当于更改了发车时间，属于“违规操作”。为了做到形式上仍然是9点发车，我们可以更改一下时钟推进的逻辑：当一个商品到达时，不要直接用它的生产时间作为当前时间，而是减去两秒，这就相当于把车上的逻辑时钟调慢了。这样一来，当9点生产的商品到达时，我们当前车上的时间是8点59分58秒，还没到发车时间；当9点零2秒生产的商品到达时，车上时间刚好是9点，这时该到的商品都到齐了，准时发车就没问题了。回到上面的例子，为了让窗口能够正确收集到迟到的数据，我们也可以等上2秒；也就是用当前已有数据的最大时间戳减去2秒，就是要插入的水位线的时间戳，这样的话，9秒的数据到来之后，事件时钟不会直接推进到9秒，而是进展到了7秒；必须等到11秒的数据到来之后，事件时钟才会进展到9秒，这时迟到数据也都已收集齐，0~9秒的窗口就可以正确计算结果了。



- 如果仔细观察就会看到，这种“等2秒”的策略其实并不能处理所有的乱序数据。比如22秒的数据到来之后，插入的水位线时间戳为20，也就是当前时钟已经推进到了20秒；对于10~20秒的窗口，这时就该关闭了。但是之后又会有17秒的迟到数据到来，它本来应该属于10~20秒窗口，现在却被遗漏丢弃了。那又该怎么办呢？
- 既然现在等2秒还是等不到17秒产生的迟到数据，那自然我们可以试着多等几秒，也就是把时钟调得更慢一些。最终的目的，就是要让窗口能够把所有迟到数据都收进来，得到正确的计算结果。对应到水位线上，其实就是要保证，当前时间已经进展到了这个时间戳，在这之后不可能再有迟到数据来了。
- 第一个水位线时间戳为7，它表示当前事件时间是7秒，7秒之前的数据都已经到齐，之后再也不会有了；同样，第二个、第三个水位线时间戳分别为12和20，表示11秒、20秒之前的数据都已经到齐，如果有对应的窗口就可以直接关闭了，统计的结果一定是正确的。这里由于水位线是周期性生成的，所以插入的位置不一定是在时间戳最大的数据后面。另外需要注意的是，这里一个窗口所收集的数据，并不是之前所有已经到达的数据。因为数据属于哪个窗口，是由数据本身的时间戳决定的，一个窗口只会收集真正属于它的那些数据。也就是说，上图中尽管水位线W(20)之前有时间戳

为22的数据到来，10~20秒的窗口中也不会收集这个数据，进行计算依然可以得到正确的结果。关于窗口的原理，我们会在后面继续展开讲解。



- 水位线特点

- 水位线是插入到数据流中的一个标记，可以认为是一个特殊的数据
- 水位线主要的内容是一个时间戳，用来表示当前事件时间的进展
- 水位线是基于数据的时间戳生成的
- 水位线的时间戳必须单调递增，以确保任务的事件时间时钟一直向前推进
- 水位线可以通过设置延迟，来保证正确处理乱序数据
- 一个水位线Watermark(t)，表示在当前流中事件时间已经到了时间戳t，这代表t之前的所有数据都到齐了，之后流中不会出现时间戳 $t' \leq t$ 的数据

12.2. 内置水位线生成器

- Flink 内置水位线生成器

- 1.10版本之前
 - AssignerWithPeriodicWatermarks
 - 周期性的生成 watermark，默认周期是200ms，也可以通过 setAutoWatermarkInterval设置周期时间
 - AssignerWithPunctuatedWatermarks
 - 阶段性的生成 watermark，即每来一条数据就生成一个wm
- 1.11版本以后
 - WatermarkStrategy
 - 单调递增策略 (forMonotonousTimestamps)
 - 固定乱序长度策略 (forBoundedOutOfOrderness)
 - 不生成策略 (noWatermarks)

12.2.1. 有序流

- 对于有序流，主要特点就是时间戳单调增长 (Monotonously Increasing Timestamps)，所以永远不会出现迟到数据的问题。
- 直接调用WatermarkStrategy.forMonotonousTimestamps()方法就可以实现。
- 直接拿当前最大的时间戳作为水位线就可以了。
- 代码实现

- ```
import com.yjxxt.util.KafkaUtil;
import org.apache.commons.lang3.RandomStringUtils;
import
org.apache.flink.api.common.eventtime.SerializableTimestampAssigner;
import org.apache.flink.api.common.eventtime.WatermarkStrategy;
import org.apache.flink.api.common.typeinfo.Types;
import org.apache.flink.api.java.tuple.Tuple3;
import org.apache.flink.streaming.api.datastream.DataStreamSource;
import
org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
```

```
import org.apache.flink.streaming.api.functions.windowing.WindowFunction;
import org.apache.flink.streaming.api.windowing.assigners.TumblingEventTimeWindows;
import org.apache.flink.streaming.api.windowing.time.Time;
import org.apache.flink.streaming.api.windowing.windows.Timewindow;
import org.apache.flink.util.collector;

import java.util.Locale;

/**
 * @Description :
 * @School:优极限学堂
 * @Official-website: http://www.yjxxt.com
 * @Teacher:李毅大帝
 * @Mail:863159469@qq.com
 */
public class Hello13WaterMarkInorder {
 public static void main(String[] args) throws Exception {

 //生产Kafka有序数据数据--模拟弹幕[用户名:消息:时间戳]
 new Thread(() -> {
 String uname =
RandomStringUtils.randomAlphabetic(8).toLowerCase(Locale.ROOT);
 for (int i = 100; i < 200; i++) {
 KafkaUtil.sendMsg("yjxxt", uname + ":" + i + ":" +
System.currentTimeMillis());
 try {
 Thread.sleep(1000);
 } catch (InterruptedException e) {
 e.printStackTrace();
 }
 }
 }).start();
 //运行环境
 StreamExecutionEnvironment environment =
StreamExecutionEnvironment.getExecutionEnvironment();
 environment.setParallelism(1);

 //读取数据源
 DataStreamSource<String> source =
environment.fromSource(KafkaUtil.getKafkaSource("yjxxt", "liyidd"),
watermarkStrategy.noWatermarks(), "Kafka Source");
 //转换数据
 source.map(line -> {
 return Tuple3.of(line.split(":")[0],
line.split(":")[1], Long.parseLong(line.split(":")[2]));
 }, Types.TUPLE(Types.STRING, Types.STRING, Types.LONG))
.assignTimestampsAndWatermarks(WatermarkStrategy.
<Tuple3<String, String, Long>>forMonotonousTimestamps()
.withTimestampAssigner(new
SerializableTimestampAssigner<Tuple3<String, String, Long>>() {
 @Override
 public long extractTimestamp(Tuple3<String,
String, Long> tuple3, long ts) {
 return tuple3.f2;
 }
})
```

```

 }))

.keyBy(tuple3 -> tuple3.f0)
.window(TumblingEventTimeWindows.of(Time.seconds(10)))
.apply(new WindowFunction<Tuple3<String, String, Long>,
String, String, Timewindow>() {
 @Override
 public void apply(String key, Timewindow window,
Iterable<Tuple3<String, String, Long>> input, collector<String> out)
throws Exception {
 StringBuffer buffer = new StringBuffer();
 buffer.append("[key]");
 for (Tuple3<String, String, Long> tuple3 : input)
{
 buffer.append("[" + tuple3.f1 + "_" +
tuple3.f2 + "]");
 }
 buffer.append("[" + window + "]");
 //返回结果
 out.collect(buffer.toString());
}
}).print();
//运行环境
environment.execute();
}
}

```

## 12.2.2. 无序流

- 由于乱序流中需要等待迟到数据到齐，所以必须设置一个固定量的延迟时间（Fixed Amount of Lateness）。
- 调用 WatermarkStrategy. forBoundedOutOfOrderness()方法就可以实现。
- 这个方法需要传入一个 maxOutOfOrderness 参数，表示“最大乱序程度”
- Tips:
  - 当程序开始时,WaterMark会被设置为Long的最小值,以保证它不会丢数据
  - 当程序关闭时,WaterMark会被设置为Long的最大值,以保证它大到足以关闭所有已经开启的窗口
- 代码实现

```

○ import com.yjxxt.util.KafkaUtil;
import org.apache.commons.lang3.RandomStringUtils;
import
org.apache.flink.api.common.eventtime.SerializableTimestampAssigner;
import org.apache.flink.api.common.eventtime.WatermarkStrategy;
import org.apache.flink.api.common.typeinfo.Types;
import org.apache.flink.api.java.tuple.Tuple3;
import org.apache.flink.streaming.api.datastream.DataStreamSource;
import
org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import
org.apache.flink.streaming.api.functions.windowing.WindowFunction;

```

```

import
org.apache.flink.streaming.api.windowing.assigners.TumblingEventTimeWindows;
import org.apache.flink.streaming.api.windowing.time.Time;
import org.apache.flink.streaming.api.windowing.windows.TimeWindow;
import org.apache.flink.util.Collector;

import java.time.Duration;
import java.util.Locale;

/**
 * @Description :
 * @School:优极限学堂
 * @Official-website: http://www.yjxxt.com
 * @Teacher:李毅大帝
 * @Mail:863159469@qq.com
 */
public class Hello14WaterMarkOutorder {
 public static void main(String[] args) throws Exception {

 //生产Kafka有序数据数据--模拟弹幕[用户名:消息:时间戳]
 new Thread(() -> {
 String uname =
RandomStringUtils.randomAlphabetic(8).toLowerCase(Locale.ROOT);
 for (int i = 100; i < 200; i++) {
 if (i % 5 != 0) {
 KafkaUtil.sendMsg("yjxxt", uname + ":" + i + ":" +
System.currentTimeMillis());
 } else {
 KafkaUtil.sendMsg("yjxxt", uname + ":" + i + ":" +
(System.currentTimeMillis() - (long) (Math.random() * 10000)));
 }
 try {
 Thread.sleep(1000);
 } catch (InterruptedException e) {
 e.printStackTrace();
 }
 }
 }).start();
 //运行环境
 StreamExecutionEnvironment environment =
StreamExecutionEnvironment.getExecutionEnvironment();
 environment.setParallelism(1);

 //读取数据源
 DataStreamSource<String> source =
environment.fromSource(KafkaUtil.getKafkaSource("yjxxt", "liyidd"),
WatermarkStrategy.noWatermarks(), "Kafka Source");
 //转换数据
 source.map(line -> {
 return Tuple3.of(line.split(":")[0],
line.split(":")[1], Long.parseLong(line.split(":")[2]));
 }, Types.TUPLE(Types.STRING, Types.STRING, Types.LONG))
.assignTimestampsAndWatermarks(WatermarkStrategy.
<Tuple3<String, String,
Long>>forBoundedOutOfOrderliness(Duration.ofSeconds(8))
.withTimestampAssigner(new
SerializableTimestampAssigner<Tuple3<String, String, Long>>() {

```

```

 @Override
 public long extractTimestamp(Tuple3<String,
String, Long> tuple3, long ts) {
 return tuple3.f2;
 }
 })
.keyBy(tuple3 -> tuple3.f0)
.window(TumblingEventTimeWindows.of(Time.seconds(10)))
.apply(new WindowFunction<Tuple3<String, String, Long>,
String, String, TimeWindow>() {
 @Override
 public void apply(String key, TimeWindow window,
Iterable<Tuple3<String, String, Long>> input, Collector<String> out)
throws Exception {
 StringBuffer buffer = new StringBuffer();
 buffer.append("[key]");
 for (Tuple3<String, String, Long> tuple3 :
input) {
 buffer.append("[" + tuple3.f1 + "_" +
tuple3.f2 + "]");
 }
 buffer.append("[" + window + "]");
 //返回结果
 out.collect(buffer.toString());
 }
}).print();
//运行环境
environment.execute();
}
}

```

## 12.3. 自定义水位线

- Flink有两种不同的生成水位线的方式：一种是周期性的（Periodic），另一种是定点式的（Punctuated）。
  - 周期性的（Periodic）：周期性调用的方法中发出水位线
    - Periodic Generator
    - 周期性生成器一般是通过 onEvent() 观察判断输入的事件，而在 onPeriodicEmit() 里发出水位线
  - 定点式的（Punctuated）：在事件触发的方法中发出水位线
    - Punctuated Generator
    - 定点式生成器会不停地检测 onEvent() 中的事件，当发现带有水位线信息的特殊事件时，就立即发出水位线。
- WatermarkGenerator 接口中有两个方法
  - onEvent()：在每个事件到来时调用
  - onPeriodicEmit()：由框架周期性调用

### 12.3.1. WatermarkGenerator

- ```

import com.yjxxt.util.KafkaUtil;
import org.apache.commons.lang3.RandomStringUtils;
```

```

import org.apache.flink.api.common.eventtime.*;
import org.apache.flink.api.common.serialization.SimpleStringSchema;
import org.apache.flink.connector.kafka.source.KafkaSource;
import
org.apache.flink.connector.kafka.source.enumerator.initializer.OffsetsInitializer;
import org.apache.flink.streaming.api.datastream.DataStreamSource;
import
org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import
org.apache.flink.streaming.api.windowing.assigners.TumblingEventTimeWindows;
import org.apache.flink.streaming.api.windowing.time.Time;

/**
 * @Description :
 * @School:优极限学堂
 * @Official-website: http://www.yjxxt.com
 * @Teacher:李毅大帝
 * @Mail:863159469@qq.com
 */
public class Hello12WatermarkGenerator {
    public static void main(String[] args) throws Exception {
        //启动一个线程专门发送消息给Kafka,这样我们才有数据消费
        new Thread(() -> {
            String uname = RandomStringUtils.randomAlphabetic(8);
            for (int i = 1000; i < 2000; i++) {
                KafkaUtil.sendMsg("yjxxt", uname + i % 2 + ":" + i + ":" +
System.currentTimeMillis());
                try {
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }).start();

        //获取环境
        StreamExecutionEnvironment environment =
StreamExecutionEnvironment.getExecutionEnvironment();
        environment.setParallelism(1);

        //设置Kafka连接
        KafkaSource<String> source = KafkaSource.<String>builder()
            .setBootstrapServers("node01:9092,node02:9092,node03:9092")
            .setTopics("yjxxt")
            .setGroupId("flink_kafkaConnector")
            .setStartingOffsets(OffsetsInitializer.latest())
            .setValueOnlyDeserializer(new SimpleStringSchema())
            .build();

        //读取数据源
        DataStreamSource<String> kafkaSource =
environment.fromSource(source, WatermarkStrategy.noWatermarks(),
"KafkaSource");
        kafkaSource.assignTimestampsAndWatermarks(new
YjxxtWatermarkStrategy()
            .keyBy(t -> t.split(":")[0])
            .window(TumblingEventTimeWindows.of(Time.seconds(5)))

```

```
        .reduce((t1, t2) -> t1 + "[" + t2.split(":")[1] + "," +
t2.split(":")[2] + "]")
        .map(t -> "[" + System.currentTimeMillis() + "][" + t + "]");
    .print("YjxxtWatermarkStrategy");
    //执行环境
    environment.execute();
}
}

class YjxxtWatermarkStrategy implements WatermarkStrategy<String> {

    @Override
    public WatermarkGenerator<String>
createWatermarkGenerator(WatermarkGeneratorSupplier.Context context) {
    return new YjxxtPeriodicGenerator();
}

    @Override
    public TimestampAssigner<String>
createTimestampAssigner(TimestampAssignerSupplier.Context context) {
    return new SerializableTimestampAssigner<String>() {
        @Override
        public long extractTimestamp(String element, long recordTimestamp) {
            return Long.valueOf(element.split(":")[2]);
        }
    };
}

/**
 * 周期型生成水位线
 */
private static class YjxxtPeriodicGenerator implements
WatermarkGenerator<String> {
    // 延迟时间
    private Long lateTime = 3000L;
    // 观察到的最大时间戳
    private Long maxTimestamp = Long.MIN_VALUE;

    @Override
    public void onEvent(String element, long eventTimestamp,
WatermarkOutput output) {
        maxTimestamp = Math.max(Long.valueOf(element.split(":")[2]),
maxTimestamp); //
    }

    @Override
    public void onPeriodicEmit(WatermarkOutput output) {
        output.emitWatermark(new Watermark(maxTimestamp - lateTime -
1L));
    }
}

/**
 * 定点型生成水位线
 */
private static class YjxxtPunctuatedGenerator implements
WatermarkGenerator<String> {
```

```

    @Override
    public void onEvent(String element, Long eventTimestamp,
    WatermarkOutput output) {
        if (Integer.parseInt(element.split(":")[1]) % 100 == 0) {
            output.emitWatermark(new
    Watermark(Long.valueOf(element.split(":")[2]) - 1L));
        }
    }

    @Override
    public void onPeriodicEmit(WatermarkOutput output) {

    }
}

```

13. 延迟数据

- 现实中很难生成一个完美的水位线，水位线就是在延迟与准确性之前做的一种权衡。如果生成的水位线过于紧迫，即水位线可能会大于后来数据的时间戳，这就意味着数据有延迟，关于延迟数据的处理，Flink提供了一些机制，具体如下：
 - allowedLateness
 - sideOutputLateData

13.1. allowedLateness

- Flink提供了WindowedStream.allowedLateness()方法来设定窗口的允许延迟。
- 正常情况下窗口触发计算完成之后就会被销毁，但是设定了允许延迟之后，窗口会等待allowedLateness的时长后开始计算并销毁。在该区间内的迟到数据仍然可以进入窗口中，并触发新的计算。
- 什么情况下数据会被丢弃或者说不会被计算？
 - 未设置allowedLateness情况下，某条数据属于某个窗口，但是watermark超过了窗口的结束时间，则该条数据会被丢弃；
 - 设置allowedLateness情况下，某条数据属于某个窗口，但是watermark超过了窗口的结束时间+延迟时间，则该条数据会被丢弃；

13.2. sideOutputLateData

- 保底方案，数据延迟严重，可以保证数据不丢失。
- 延迟的数据通过outputTag输出，必须要事件时间大于watermark + allowed lateness，数据才会存储在outputTag中。
- 代码实现

```

    import com.yjxxt.util.KafkaUtil;
    import org.apache.commons.lang3.RandomStringUtils;
    import
    org.apache.flink.api.common.eventtime.SerializableTimestampAssigner;
    import org.apache.flink.api.common.eventtime.WatermarkStrategy;
    import org.apache.flink.api.common.typeinfo.Types;
    import org.apache.flink.api.java.tuple.Tuple3;

```

```
import org.apache.flink.streaming.api.datastream.DataStreamSource;
import
org.apache.flink.streaming.api.datastream.SingleOutputStreamOperator;
import
org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import
org.apache.flink.streaming.api.functions.windowing.WindowFunction;
import
org.apache.flink.streaming.api.windowing.assigners.TumblingEventTimeWindows;
import org.apache.flink.streaming.api.windowing.time.Time;
import org.apache.flink.streaming.api.windowing.windows.TimeWindow;
import org.apache.flink.util.Collector;
import org.apache.flink.util.OutputTag;

import java.time.Duration;
import java.util.Locale;

/**
 * @Description :
 * @School:优极限学堂
 * @Official-website: http://www.yjxxt.com
 * @Teacher:李毅大帝
 * @Mail:863159469@qq.com
 */
public class Hello18WaterMarkLateSide {
    public static void main(String[] args) throws Exception {

        //生产Kafka有序数据数据--模拟弹幕[用户名:消息:时间戳]
        new Thread(() -> {
            String uname =
RandomStringUtils.randomAlphabetic(8).toLowerCase(Locale.ROOT);
            for (int i = 100; i < 200; i++) {
                if (i % 10 == 0) {
                    KafkaUtil.sendMsg("yjxxt", uname + ":" + i + ":" +
(System.currentTimeMillis() - 15000L));
                } else if (i % 5 == 0) {
                    KafkaUtil.sendMsg("yjxxt", uname + ":" + i + ":" +
(System.currentTimeMillis() - 3000L));
                } else {
                    KafkaUtil.sendMsg("yjxxt", uname + ":" + i + ":" +
System.currentTimeMillis());
                }
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }).start();
        //运行环境
        StreamExecutionEnvironment environment =
StreamExecutionEnvironment.getExecutionEnvironment();
        environment.setParallelism(1);

        //读取数据源
```

```

        DataStreamSource<String> source =
environment.fromSource(kafkaUtil.getKafkaSource("yjxxt", "liyidd"),
watermarkStrategy.noWatermarks(), "Kafka Source");
        //声明侧输出对象
        OutputTag<Tuple3<String, String, Long>> outputTag = new
OutputTag<>("w10d115") {
};

        //转换数据
        SingleOutputStreamOperator<String> streamOperator =
source.map(line -> {
    return Tuple3.of(line.split(":")[0],
line.split(":")[1], Long.parseLong(line.split(":")[2]));
}, Types.TUPLE(Types.STRING, Types.STRING, Types.LONG))
.assignTimestampsAndWatermarks(watermarkStrategy.
<Tuple3<String, String,
Long>>forBoundedOutOfOrderness(Duration.ofSeconds(1)))
.withTimestampAssigner(new
SerializableTimestampAssigner<Tuple3<String, String, Long>>() {
    @Override
    public long extractTimestamp(Tuple3<String,
String, Long> tuple3, long ts) {
        return tuple3.f2;
    }
})
.keyBy(tuple3 -> tuple3.f0)
.window(TumblingEventTimeWindows.of(Time.seconds(10)))
.allowedLateness(Time.seconds(5))
.sideOutputLateData(outputTag)
.apply(new WindowFunction<Tuple3<String, String, Long>,
String, String, TimeWindow>() {
    @Override
    public void apply(String key, TimeWindow window,
Iterable<Tuple3<String, String, Long>> input, Collector<String> out)
throws Exception {
        StringBuffer buffer = new StringBuffer();
        buffer.append("[ " + key + " ]");
        for (Tuple3<String, String, Long> tuple3 :
input) {
            buffer.append("[ " + tuple3.f1 + " _ "
tuple3.f2 + " ]");
        }
        buffer.append("[ " + window + " ]");
        //返回结果
        out.collect(buffer.toString());
    }
});
//主流数据
streamOperator.print("Main:");

        //侧输出数据
streamOperator.getSideOutput(outputTag).print("Side:");

        //运行环境
environment.execute();
}
}

```

14. Trigger与Evictor

- window operator 包含四个组件，包括 window assigner, trigger , evictor, window process。
 - window assigner 指明数据流中的数据属于哪个window
 - trigger 指明在哪些条件下触发window计算，基于处理数据时的时间以及事件的特定属性
 - evictor 可选组件，在window执行计算前或后，将window中的数据移除，如使用 globalWindow时，由于该window的默认trigger为永不触发，所以既需要实现自定义 trigger，也需要实现evictor，移除部分已经计算完毕的数据。
 - window process flink默认提供的有 ReduceFunction,AggragateFunction.还可以自定义实现 windowProcessFunction

14.1. Trigger

- 窗口触发器，决定了窗口什么时候使用窗口函数处理窗口内元素。每个窗口分配器都带有一个默认的触发器。
- Trigger 决定了一个窗口（由 `window assigner` 定义）何时可以被 `window function` 处理。每个 `windowAssigner` 都有一个默认的 `Trigger`。
- 如果默认 trigger 无法满足你的需要，可以在 `trigger(...)` 调用中指定自定义的 trigger。
- Trigger 接口提供了五个方法来响应不同的事件：
 - `onElement()` 方法在每个元素被加入窗口时调用。
 - `onEventTime()` 方法在注册的 event-time timer 触发时调用。
 - `onProcessingTime()` 方法在注册的 processing-time timer 触发时调用。
 - `onMerge()` 方法与有状态的 trigger 相关。该方法会在两个窗口合并时，将窗口对应 trigger 的状态进行合并，比如使用会话窗口时。
 - `clear()` 方法处理在对应窗口被移除时所需的逻辑。
- 前三个方法通过返回 `TriggerResult` 来决定 trigger 如何应对到达窗口的事件。应对方案有以下几种：

```
◦ public enum TriggerResult {  
    // 表示对窗口不执行任何操作。即不触发窗口计算，也不删除元素。  
    CONTINUE(false, false),  
    // 触发窗口计算，输出结果，然后将窗口中的数据和窗口进行清除。  
    FIRE_AND_PURGE(true, true),  
    // 触发窗口计算，但是保留窗口元素  
    FIRE(true, false),  
    // 不触发窗口计算，丢弃窗口，并且删除窗口的元素。  
    PURGE(false, true);  
  
    private final boolean fire;  
    private final boolean purge;  
  
    private TriggerResult(boolean fire, boolean purge) {  
        this.purge = purge;  
        this.fire = fire;  
    }  
}
```

- `windowAssigner` 默认的 `Trigger` 足以应付诸多情况。

- EventTimeTrigger: 通过对比EventTime和窗口的Endtime确定是否触发窗口计算，如果EventTime大于Window EndTime则触发，否则不触发，窗口将继续等待。
- ProcessTimeTrigger: 通过对比ProcessTime和窗口EndTime确定是否触发窗口，如果ProcessTime大于EndTime则触发计算，否则窗口继续等待。
- ContinuousEventTimeTrigger: 根据间隔时间周期性触发窗口或者Window的结束时间小于当前EndTime触发窗口计算。
- ContinuousProcessingTimeTrigger: 根据间隔时间周期性触发窗口或者Window的结束时间小于当前ProcessTime触发窗口计算。
- CountTrigger: 根据接入数据量是否超过设定的阈值判断是否触发窗口计算。
- DeltaTrigger: 根据接入数据计算出来的Delta指标是否超过指定的Threshold去判断是否触发窗口计算。
- PurgingTrigger: 可以将任意触发器作为参数转换为Purge类型的触发器，计算完成后数据将被清理。
- NeverTrigger: 任何时候都不触发窗口计算

| 内置Trigger | 说明 |
|---------------------------------|-------------------------------------------------------------|
| ProcessingTimeTrigger | 一次触发, machine time大于窗口结束时间时触发 |
| EventTimeTrigger | 一次触发, watermark大于窗口结束时间时触发 |
| ContinuousProcessingTimeTrigger | 多次触发, 基于processing time的固定时间间隔 |
| ◦ ContinuousEventTimeTrigger | 多次触发, 基于event time的固定时间间隔 |
| CountTrigger | 多次触发, 基于element的固定条数 |
| DeltaTrigger | 多次触发, 当前element与上次触发trigger的element做delta计算, 超过threshold时触发 |
| PurgingTrigger | trigger wrapper, 当nested trigger触发时, 额外会清理窗口当前的中间状态 |

14.2. Evictor

- Flink 窗口模型还允许在窗口分配器和触发器之外指定一个可选的驱逐器(Evictor)
- 驱逐器能够在触发器触发之后，窗口函数使用之前或之后从窗口中清除元素。
- evictBefore()在窗口函数之前使用。而 evictAfter() 在窗口函数之后使用。在使用窗口函数之前被逐出的元素将不被处理。
- Flink带有三种内置驱逐器:
 - CountEvictor: 数量剔除器。在 Window 中保留指定数量的元素，并从窗口头部开始丢弃其余元素。
 - DeltaEvictor: 阈值剔除器。计算 Window 中最后一个元素与其余每个元素之间的增量，丢弃增量大于或等于阈值的元素。
 - TimeEvictor: 时间剔除器。保留 Window 中最近一段时间内的元素，并丢弃其余元素。
- 默认情况下，所有内置的驱逐器在窗口函数之前使用。指定驱逐器可以避免预聚合(pre-aggregation)，因为窗口内所有元素必须在窗口计算之前传递给驱逐器。
- Flink 不保证窗口内元素的顺序。这意味着虽然驱逐器可以从窗口开头移除元素，但这些元素不一定是先到的还是后到的。
- 代码实现

```

◦ import org.apache.flink.api.common.typeinfo.Types;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.streaming.api.datastream.DataStreamSource;
import
org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import
org.apache.flink.streaming.api.windowing.assigners.GlobalWindows;
import org.apache.flink.streaming.api.windowing.evictors.CountEvictor;
```

```

import org.apache.flink.streaming.api.windowing.triggers.CountTrigger;

/**
 * @Description :
 * @School:优极限学堂
 * @Official-website: http://www.yjxxt.com
 * @Teacher:李毅大帝
 * @Mail:863159469@qq.com
 */
public class Hello19TriggerAndEvictor {
    public static void main(String[] args) throws Exception {
        //运行环境
        StreamExecutionEnvironment environment =
StreamExecutionEnvironment.getExecutionEnvironment();
        //获取数据源-admin:3
        DataStreamSource<String> source =
environment.socketTextStream("localhost", 19523);
        //Timewindow--Sliding
        source.map(word -> Tuple2.of(word.split(":")[0],
Integer.parseInt(word.split(":")[1])), Types.TUPLE(Types.STRING,
Types.INT))
            .keyBy(tuple2 -> tuple2.f0)
            .window(GlobalWindows.create())
            .trigger(CountTrigger.of(10))
            .evictor(CountEvictor.of(10))
            .reduce((t1, t2) -> {
                t1.f1 = t1.f1 + t2.f1;
                return t1;
            })
            .print("Timewindow--Sliding:").setParallelism(1);

        //运行环境
        environment.execute();
    }
}

```

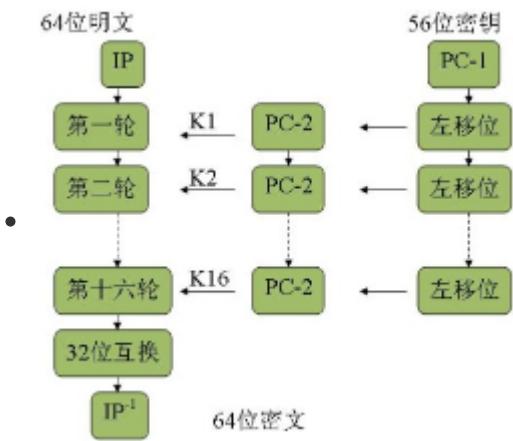
15. 附录:

15.1. DESUtil

15.1.1. 算法简介

- DES(Data Encryption Standard)是目前最为流行的加密算法之一。DES是对称的，也就是说它使用同一个密钥来加密和解密数据。
- DES算法具体通过对明文进行一系列的排列和替换操作来将其加密。过程的关键就是从给定的初始密钥中得到16个子密钥的函数。要加密一组明文，每个子密钥按照顺序（1-16）以一系列的位操作施加于数据上，每个子密钥一次，一共重复16次。每一次迭代称之为一轮。要对密文进行解密可以采用同样的步骤，只是子密钥是按照逆向的顺序（16-1）对密文进行处理。

15.1.2. 加密原理



- DES 使用一个 56 位的密钥以及附加的 8 位奇偶校验位，产生最大 64 位的分组大小。这是一个迭代的分组密码，使用称为 Feistel 的技术，其中将加密的文本块分成两半。使用子密钥对其中一半应用循环功能，然后将输出与另一半进行“异或”运算；接着交换这两半，这一过程会继续下去，但最后一个循环不交换。DES 使用 16 个循环，使用异或，置换，代换，移位操作四种基本运算。
- DES 的常见变体是三重 DES，使用 168 位的密钥对资料进行三次加密的一种机制；它通常（但非始终）提供极其强大的安全性。如果三个 56 位的子元素都相同，则三重 DES 向后兼容 DES。

15.1.3. 破解原理

- 攻击 DES 的主要形式被称为蛮力的或彻底密钥搜索，即重复尝试各种密钥直到有一个符合为止。如果 DES 使用 56 位的密钥，则可能的密钥数量是 2^{56} 。随着计算机系统能力的不断发展，DES 的安全性比它刚出现时会弱得多，然而从非关键性质的实际出发，仍可以认为它是足够的。不过，DES 仅用于旧系统的鉴定，而更多地选择新的加密标准 — 高级加密标准 (Advanced Encryption Standard, AES)。
- 新的分析方法有差分分析法和线性分析法两种

15.1.4. 代码实现

```

• import javax.crypto.Cipher;
import javax.crypto.SecretKeyFactory;
import javax.crypto.spec.DESKeySpec;
import javax.crypto.spec.IvParameterSpec;
import java.security.Key;
import java.util.Base64;

/**
 * Des 加密工具类
 */
public class DESUtil {
    /**
     * 偏移变量，固定占8位字节
     */
    private final static String IV_PARAMETER = "12345678";
    /**
     * 密钥算法
     */
    private static final String ALGORITHM = "DES";
    /**
     * 加密/解密算法-工作模式-填充模式
     */
    private static final String CIPHER_ALGORITHM = "DES/CBC/PKCS5Padding";
    /**
     * 默认编码
     */
}

```

```
private static final String CHARSET = "utf-8";

/**
 * 生成key
 *
 * @param password
 * @return
 * @throws Exception
 */
private static Key generateKey(String password) throws Exception {
    DESKeySpec dks = new DESKeySpec(password.getBytes(CHARSET));
    SecretKeyFactory keyFactory =
SecretKeyFactory.getInstance(ALGORITHM);
    return keyFactory.generateSecret(dks);
}

/**
 * DES加密字符串
 *
 * @param password 加密密码，长度不能够小于8位
 * @param data      待加密字符串
 * @return 加密后内容
 */
public static String encrypt(String password, String data) {
    if (password == null || password.length() < 8) {
        throw new RuntimeException("加密失败，key不能小于8位");
    }
    if (data == null)
        return null;
    try {
        Key secretKey = generateKey(password);
        Cipher cipher = Cipher.getInstance(CIPHER_ALGORITHM);
        IvParameterSpec iv = new
IvParameterSpec(IV_PARAMETER.getBytes(CHARSET));
        cipher.init(Cipher.ENCRYPT_MODE, secretKey, iv);
        byte[] bytes = cipher.doFinal(data.getBytes(CHARSET));

        //JDK1.8及以上可直接使用Base64，JDK1.7及以下可以使用BASE64Encoder
        //Android平台可以使用android.util.Base64
        return new String(Base64.getEncoder().encode(bytes));

    } catch (Exception e) {
        e.printStackTrace();
        return data;
    }
}

/**
 * DES解密字符串
 *
 * @param password 解密密码，长度不能够小于8位
 * @param data      待解密字符串
 * @return 解密后内容
 */
public static String decrypt(String password, String data) {
    if (password == null || password.length() < 8) {
        throw new RuntimeException("加密失败，key不能小于8位");
    }
}
```

```
    if (data == null)
        return null;
    try {
        Key secretKey = generateKey(password);
        Cipher cipher = Cipher.getInstance(CIPHER_ALGORITHM);
        IvParameterSpec iv = new
IvParameterSpec(IV_PARAMETER.getBytes(CHARSET));
        cipher.init(Cipher.DECRYPT_MODE, secretKey, iv);
        return new
String(cipher.doFinal(Base64.getDecoder().decode(data.getBytes(CHARSET))), CHARSET);
    } catch (Exception e) {
        e.printStackTrace();
        return data;
    }
}

public static void main(String[] args) {
    //加密数据
    System.out.println(DESUtil.encrypt("yjxxt0523", "李毅老师好帅"));

    //解密数据
    System.out.println(DESUtil.decrypt("yjxxt0523",
"IuJ6j31kaCjkGZpPuPrw47L9fpAkDQcL"));

}
```