- **导入依赖**

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>sz.base.flink</groupId>
    <artifactId>flink-base-sz28</artifactId>
    <packaging>pom</packaging>
    <version>1.0-SNAPSHOT</version>
    <modules>
        <module>day01_wordcount</module>
    </modules>


    <repositories>
        <repository>
            <releases>
                <enabled>true</enabled>
                <updatePolicy>never</updatePolicy>
                <checksumPolicy>fail</checksumPolicy>
            </releases>
            <snapshots>
                <enabled>false</enabled>
                <updatePolicy>always</updatePolicy>
                <checksumPolicy>warn</checksumPolicy>
            </snapshots>
            <id>cdh.repo</id>
            <name>Cloudera Repositories</name>
            <url>https://repository.cloudera.com/artifactory/cloudera-repos</url>
            <layout>default</layout>
        </repository>

        <repository>
            <id>spring</id>
            <url>https://repo.spring.io/plugins-release/</url>
            <releases>
                <updatePolicy>always</updatePolicy>
            </releases>
        </repository>
    </repositories>

    <properties>
        <flink.version>1.13.1</flink.version>
        <java.version>1.8</java.version>
        <scala.binary.version>2.11</scala.binary.version>
        <hadoop.version>3.3.0</hadoop.version>
        <hbase.version>2.0.0</hbase.version>
        <zkclient.version>0.8</zkclient.version>
        <hive.version>2.1.1</hive.version>
        <mysql.version>5.1.48</mysql.version>
        <log4j.version>1.7.32</log4j.version>
        <logback.version>1.2.6</logback.version>
    </properties>


    <dependencies>
        <!-- Apache Flink dependencies -->
        <!-- These dependencies are provided, because they should not be packaged into
the JAR file. -->
        <!--<dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-java</artifactId>
            <version>${flink.version}</version>
            &lt;!&ndash;<scope>provided</scope>&ndash;&gt;
        </dependency>-->
        <!-- https://mvnrepository.com/artifact/junit/junit -->
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.13.2</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-streaming-scala_${scala.binary.version}</artifactId>
            <version>${flink.version}</version>
            <!--<scope>compile</scope>-->
        </dependency>

        <dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-runtime-web_${scala.binary.version}</artifactId>
            <version>${flink.version}</version>
        </dependency>

        <!-- https://mvnrepository.com/artifact/org.apache.flink/flink-clients -->
        <dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-clients_${scala.binary.version}</artifactId>
            <version>${flink.version}</version>
        </dependency>

        <!-- https://mvnrepository.com/artifact/org.apache.flink/flink-table -->
        <dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-table-planner_2.11</artifactId>
            <version>${flink.version}</version>
        </dependency>

        <dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-table-planner-blink_2.11</artifactId>
            <version>${flink.version}</version>
        </dependency>

        <dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-table-runtime-blink_2.11</artifactId>
            <version>${flink.version}</version>
        </dependency>

        <dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-table-api-scala-bridge_2.11</artifactId>
```

```xml
                <version>${flink.version}</version>
            </dependency>


            <dependency>
                <groupId>org.apache.flink</groupId>
                <artifactId>flink-java</artifactId>
                <version>${flink.version}</version>
            </dependency>
            <!--kafka-->
            <dependency>
                <groupId>org.apache.flink</groupId>
                <artifactId>flink-connector-kafka_2.11</artifactId>
                <version>${flink.version}</version>
            </dependency>
            <dependency>
                <groupId>org.apache.flink</groupId>
                <artifactId>flink-sql-connector-kafka_2.11</artifactId>
                <version>${flink.version}</version>
            </dependency>

            <dependency>
                <groupId>org.apache.flink</groupId>
                <artifactId>flink-queryable-state-runtime_2.11</artifactId>
                <version>${flink.version}</version>
            </dependency>

            <!-- https://mvnrepository.com/artifact/org.apache.flink/flink-statebackend-rocksdb -->
            <dependency>
                <groupId>org.apache.flink</groupId>
                <artifactId>flink-statebackend-rocksdb_2.11</artifactId>
                <version>${flink.version}</version>
            </dependency>

            <dependency>
                <groupId>org.apache.kafka</groupId>
                <artifactId>kafka-clients</artifactId>
                <version>1.0.0</version>

```

```xml
        </dependency>
        <!--es6-->
        <dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-connector-elasticsearch6_${scala.binary.version}
</artifactId>
            <version>${flink.version}</version>
        </dependency>

        <!--flink-jdbc-->
        <dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-connector-jdbc_2.11</artifactId>
            <version>${flink.version}</version>
        </dependency>

        <!--flink-hbase-->

        <dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-connector-hbase-2.2_2.11</artifactId>
            <version>${flink.version}</version>
            <exclusions>
                <exclusion>
                    <artifactId>slf4j-api</artifactId>
                    <groupId>org.slf4j</groupId>
                </exclusion>
            </exclusions>
        </dependency>


        <!--hadoop-->
        <!-- https://mvnrepository.com/artifact/org.apache.hadoop/hadoop-common -->
        <dependency>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-common</artifactId>
            <version>${hadoop.version}</version>
            <exclusions>
                <exclusion>
                    <artifactId>slf4j-api</artifactId>
                    <groupId>org.slf4j</groupId>
                </exclusion>
                <exclusion>
                    <artifactId>slf4j-log4j12</artifactId>
                    <groupId>org.slf4j</groupId>
                </exclusion>
            </exclusions>
        </dependency>
        <dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-hadoop-compatibility_2.11</artifactId>
            <version>${flink.version}</version>
        </dependency>
        <dependency>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-client</artifactId>
            <version>${hadoop.version}</version>
            <exclusions>
                <exclusion>
                    <artifactId>slf4j-api</artifactId>
                    <groupId>org.slf4j</groupId>
                </exclusion>
            </exclusions>
        </dependency>

        <dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-shaded-hadoop-2-uber</artifactId>
            <version>2.7.5-10.0</version>
            <exclusions>
                <exclusion>
                    <artifactId>slf4j-log4j12</artifactId>
                    <groupId>org.slf4j</groupId>
                </exclusion>
            </exclusions>
        </dependency>

        <dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-csv</artifactId>
```

```xml
            <version>${flink.version}</version>
        </dependency>

        <!--flink-hbase-->
        <dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-json</artifactId>
            <version>${flink.version}</version>
            <!--<scope>test</scope>-->
        </dependency>

        <dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-runtime_${scala.binary.version}</artifactId>
            <version>${flink.version}</version>
            <!--<scope>test</scope>-->
        </dependency>

        <!-- log4j日志 start-->
        <dependency>
            <groupId>ch.qos.logback</groupId>
            <artifactId>logback-core</artifactId>
            <version>${logback.version}</version>
        </dependency>
        <dependency>
            <groupId>ch.qos.logback</groupId>
            <artifactId>logback-classic</artifactId>
            <version>${logback.version}</version>
        </dependency>
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>log4j-over-slf4j</artifactId>
            <version>${log4j.version}</version>
            <exclusions>
                <exclusion>
                    <artifactId>slf4j-api</artifactId>
                    <groupId>org.slf4j</groupId>
                </exclusion>
            </exclusions>
        </dependency>

        <!-- json -->
        <dependency>
            <groupId>com.alibaba</groupId>
            <artifactId>fastjson</artifactId>
            <version>1.2.5</version>
        </dependency>

        <!-- On hive -->
        <!-- Flink Dependency -->

        <dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-connector-hive_${scala.binary.version}</artifactId>
            <version>${flink.version}</version>
        </dependency>
        <!-- Hive Dependency -->
        <dependency>
            <groupId>org.apache.hive</groupId>
            <artifactId>hive-exec</artifactId>
            <version>${hive.version}</version>
            <exclusions>
                <exclusion>
                    <artifactId>log4j-slf4j-impl</artifactId>
                    <groupId>org.apache.logging.log4j</groupId>
                </exclusion>
            </exclusions>
        </dependency>

        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <version>1.18.2</version>
            <scope>provided</scope>
        </dependency>

        <!-- mysql 连接驱动 -->
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>${mysql.version}</version>
        </dependency>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>RELEASE</version>
            <scope>compile</scope>
        </dependency>

        <!-- redis客户端 -->
        <dependency>
            <groupId>redis.clients</groupId>
            <artifactId>jedis</artifactId>
            <version>2.9.0</version>
        </dependency>
        <!-- flink 连接 redis 的 connector -->
        <dependency>
            <groupId>org.apache.bahir</groupId>
            <artifactId>flink-connector-redis_2.11</artifactId>
            <version>1.0</version>
        </dependency>
    </dependencies>

    <build>
        <!-- 默认加载此目录，作为source目录-->
        <sourceDirectory>src/main/java</sourceDirectory>
        <plugins>
            <!-- java编译插件 -->
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.6.1</version>
                <configuration>
                    <source>${java.version}</source>
                    <target>${java.version}</target>
                    <encoding>UTF-8</encoding>
                </configuration>
            </plugin>
            <!-- scala编译插件 -->
            <plugin>
                <groupId>net.alchim31.maven</groupId>
                <artifactId>scala-maven-plugin</artifactId>
                <version>4.0.2</version>
                <executions>
                    <execution>
                        <id>compile-scala</id>
                        <phase>compile</phase>
                        <goals>
                            <goal>add-source</goal>
                            <goal>compile</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
            <!-- 打jar包插件(会包含所有依赖) -->
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-assembly-plugin</artifactId>
                <version>2.6</version>
                <configuration>
                    <descriptorRefs>
                        <descriptorRef>jar-with-dependencies</descriptorRef>
                    </descriptorRefs>
                    <archive>
                        <manifest>
                            <!-- 可以设置jar包的入口类(可选) -->

<mainClass>cn.itcast.flink.checkpoint.SavepointDemo</mainClass>
                        </manifest>
                    </archive>
                </configuration>
                <executions>
                    <execution>
                        <id>make-assembly</id>
                        <phase>package</phase>
                        <goals>
                            <goal>single</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

- **ExecutionEnvironment**

```java
package sz.base.flink.wordcount;


import org.apache.flink.api.common.functions.FlatMapFunction;
import org.apache.flink.api.common.functions.MapFunction;
import org.apache.flink.api.java.ExecutionEnvironment;
import org.apache.flink.api.java.operators.AggregateOperator;
import org.apache.flink.api.java.operators.DataSource;
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;

/**
 * 编写flink程序，读取文件中的字符串，并以空格进行单词拆分打印
 * 1.获取批的执行环境
 * 2.读取文件数据
 * 3.将读取到文件数据进行拆分 hello,world,flink,hadoop,hello
 * 4.将拆分的单词转换成(hello,1)(world,)
 * 5.需要根据单词进行分组
 * 6.根据组内进行统计求和
 * 7.将结果打印输出到控制台
 */
public class WordcountBatchDemo {
    public static void main(String[] args) throws Exception {

//          * 1.获取批的执行环境
        ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();
//          * 2.读取文件数据
        DataSource<String> source = env.readTextFile("input/words.txt");
//          * 3.将读取到文件数据进行拆分 hello,world,flink,hadoop,hello
        source.flatMap(new FlatMapFunction<String, String>() {
            @Override
            public void flatMap(String s, Collector<String> collector) throws Exception
{
                String[] words = s.split(",");
                for (String word : words) {
                    collector.collect(word);
                }
            }
            //          * 4.将拆分的单词转换成(hello,1)(world,)
        }).map(new MapFunction<String, Tuple2<String, Integer>>() {
            @Override
            public Tuple2<String, Integer> map(String s) throws Exception {
                return Tuple2.of(s, 1);
            }
            //          * 5.需要根据单词进行分组
//          * 6.根据组内进行统计求和
        }).groupBy(0).sum(1).print();
//          * 7.将结果打印输出到控制台
        env.execute();

    }
}

```

- **StreamExecutionEnvironment**

```java
package sz.base.flink.wordcount;

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.environment.StreamExecutionEnvironment;
import org.apache.flink.util.Collector;

public class WordcountStreamDemo {
    public static void main(String[] args) throws Exception {
        //1.创建流执行环境
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
        //2.读取socket数据源
        DataStreamSource<String> node1 = env.socketTextStream("192.168.88.161", 9999);
        //3.对读取进行来的数据进行flatMap拆分
        node1.flatMap(new FlatMapFunction<String, String>() {
            @Override
            public void flatMap(String s, Collector<String> collector) throws Exception
{
                String[] words = s.split(" ");
                for (String word : words) {
                    collector.collect(word);
                }
            }
        }).map(new MapFunction<String, Tuple2<String, Integer>>() {
            @Override
            public Tuple2<String, Integer> map(String s) throws Exception {
                return Tuple2.of(s, 1);
            }
        }).keyBy(new KeySelector<Tuple2<String, Integer>, String>() {
            @Override
            public String getKey(Tuple2<String, Integer> stringIntegerTuple2) throws
Exception {
                return stringIntegerTuple2.f0;
            }
        }).sum(1).print();
        env.execute();
    }
}

```

- ## StreamLambda

```java
package sz.base.flink.wordcount;

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.util.Collector;


import java.util.Arrays;

public class WordCountLambdaDemo {
    public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
        DataStreamSource<String> node1 = env.socketTextStream("node1", 9999);
        node1.flatMap((String var1, Collector<String> var2) ->
                Arrays.stream(var1.split(" ")).forEach(var2::collect)
        ).returns(Types.STRING).map(t -> Tuple2.of(t,
1)).returns(Types.TUPLE(Types.STRING, Types.INT)).keyBy(k -> k.f0).sum(1).print();
        env.execute();
    }
}

```

- **source集合**

```java
package sz.base.flink.source;

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

import java.util.ArrayList;
import java.util.List;

public class ElementListDemo2 {
    public static void main(String[] args) throws Exception {
// * 1.创建流执行环境
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(1);
//* 2.创建本地的数据流
//        DataStreamSource<Integer> source = env.fromElements(1, 2, 3, 4, 5, 6, 7, 8,
9,10);
        //从collection中获取集合
        List<Student> studentList = new ArrayList<>();
        studentList.add(new Student("zhangsan", 22));
        studentList.add(new Student("lisi", 24));
        DataStreamSource<Student> source = env.fromCollection(studentList);
        //接受一个data
        DataStreamSource<Integer> source1 = env.fromElements(1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 10);
        //接受一个范围
        DataStreamSource<Long> source2 = env.fromSequence(1, 100);
//     * 3.打印输出数据流
        source.printToErr();
//     * 4.执行流环境
        env.execute();
    }

    public static class Student {
        private String name;
        private int age;

        public Student(String name, int age) {
            this.name = name;
            this.age = age;
        }

        public String getName() {
            return name;
        }

        public void setName(String name) {
            this.name = name;
        }

        public int getAge() {
            return age;
        }

        @Override
        public String toString() {
            return"当前学生: "+ this.name+" 的年龄为: "+this.age;
        }

        public void setAge(int age) {
            this.age = age;
```

```java




            }
        }
}

```

- ## source读取文件

```java
package sz.base.flink.file;

import org.apache.flink.api.java.io.TextInputFormat;
import org.apache.flink.streaming.api.datastream.DataStreamSource;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.functions.source.FileProcessingMode;

/**
 * 读取文件中的数据
 * 1.读取一个普通的文本文件（once），静态文件
 * 2.读取一个文本文件（once），readFile，静态文件
 * 3.读取一个文本文件，每5s钟重新加载一下文件读取，读取流文本数据
 * 开发步骤:
 * 1.创建流执行环境
 * 2.设置并行度等参数
 * 3.读取文本文件的数据源
 * 4.打印输出文本文件的内容
 * 5.执行流环境
 */
public class TextFile {
    public static void main(String[] args) throws Exception {
//          * 1.创建流执行环境
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
//                 * 2.设置并行度等参数
        env.setParallelism(1);
//                 * 3.读取文本文件的数据源
        //用于批处理任务，数据时静态的，有界的
//          DataStreamSource<String> source = env.readTextFile("input/abc.csv");
```

```
        //读取一个文本文件，readFile静态文件
//        DataStreamSource<String> source = env.readFile(new TextInputFormat(null), "input/abc.csv");
        //每隔5s读取一次,会把所有文件重新读取一次，只有文件内容变化才会重新打印输出一次,
        DataStreamSource<String> source = env.readFile(new TextInputFormat(null), "input/abc.csv", FileProcessingMode.PROCESS_CONTINUOUSLY, 5000L);
        //只读取一次
        DataStreamSource<String> source1 = env.readFile(new TextInputFormat(null), "input/abc.csv", FileProcessingMode.PROCESS_ONCE, 5000L);
//                  * 4.打印输出文本文件的内容
        source.printToErr();
//                  * 5.执行流环境
        env.execute("读取文本文件的 job");
    }
}

```

ParallelSourceFunction 接口案例

- **自定义source**

```
package sz.base.flink.cs;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.functions.source.SourceFunction;

import java.text.SimpleDateFormat;
import java.util.Random;
import java.util.UUID;

/**
 * 每1s钟随机生成一条订单信息（订单ID、用户ID、订单金额、时间戳）
 * 开发步骤:
 * 1.创建流执行环境
 * 2.实现一个SourceFunction类，重写run方法，实现每一秒钟打印一条数据
 * 3.生成订单，封装一个Order的类
 * 4.将这个自定义生成的订单数据流打印输出
```

```java
 * 5.执行流环境
 */
public class CustomOrder1 {
    public static void main(String[] args) throws Exception {
//          * 1.创建流执行环境
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(1);
//                 * 2.实现一个SourceFunction类，重写run方法，实现每一秒钟打印一条数据
        env.addSource(new CustomSource()).printToErr();
//                 * 3.生成订单，封装一个Order的类
//                 * 4.将这个自定义生成的订单数据流打印输出
//                 * 5.执行流环境
        env.execute();
    }
    public static class CustomSource implements SourceFunction<Order>{
            //定义一个标记
            private boolean isRuning=true;
            Random rm=new Random();
            //定义一个格式化工具
            SimpleDateFormat sdf= new SimpleDateFormat("yyyy-MM-dd HH:mm:ss.sss");
            /**
             * 生成自定义数据源的业务逻辑
             * @param sourceContext 收集器
             * @throws Exception
             */
            @Override
            public void run(SourceContext<Order> sourceContext) throws Exception {
                //每1s钟随机生成一条订单信息（订单ID、用户ID、订单金额、时间戳）
                while (isRuning){
                    Order order=new Order(
                            UUID.randomUUID().toString(),
                            rm.nextInt(3),
                            rm.nextInt(101),
                            System.currentTimeMillis(),
                            sdf.format(System.currentTimeMillis())
                    );
                    sourceContext.collect(order);
                    Thread.sleep(1000);
                }
            }

            /**
             * 用户取消生成自定义数据源的方式
             */
            @Override
            public void cancel() {
                isRuning=false;
            }
        }


    @Data
    @AllArgsConstructor
    @NoArgsConstructor
    public static class Order{
        //随机生成订单ID（UUID）
        private String oid;
        //随机生成用户ID（0-2）
        private int uid;
        //随机生成订单金额（0-100）
        private int money;
        //时间戳为当前系统时间
        private long timestamp;
        //当前时间
        private String datetime;

        }
    }


```

RichParallelSourceFunction案例

- **读取MySQL**

- Rich 是富函数继承了 AbstractRichFunciton，实现了
- 生命周期的 open 和 close 方法
  a. open 方法，用于实现当前生成的初始化条件

b. close 方法，用于生成数据结束的收尾工作

c. getRuntimeContext 方法，用于获取当前的程序的上下文对象（参数、环境变量、状态、累加器等）

```java
package sz.base.flink.cs;

import lombok.AllArgsConstructor;
import lombok.Data;
import org.apache.flink.configuration.Configuration;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.functions.source.RichParallelSourceFunction;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;

/**
 * 读取数据库中的数据表数据并打印输出
 * 开发步骤：
 * 1.获取流执行环境
 * 2.设置参数设置并行度
 * 3.读取自定义数据源RichSourceFunction
 * RichSourceFunction与SourceFunction多了个AbstractRichFunction抽象类，多了两个方法：
 * 称为生命周期方法
 * open（）：开启链接
 * close（）：关闭链接
 * 4.自定义一个类user用于接收返回的数据
 * 5.打印输出User对象
 * 6.执行流环境
 */
public class UserSource {
    public static void main(String[] args) throws Exception {
        // 1.
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(1);
        env.addSource(new RichParallelSourceFunction<User>() {
            private Connection conn=null;
            private Statement statement=null;
            private boolean isRunning=true;
            /**
             * 生命周期的开始，再次方法中注意用于定义初始化的操作
             * 初始化的动作就是连接数据库，就连接一次
             * @param parameters
             * @throws Exception
             */
            @Override
            public void open(Configuration parameters) throws Exception {
                //1.设置连接MySQL驱动
                Class.forName("com.mysql.jdbc.Driver");
                //2.获取MySQL的连接
                conn=DriverManager.getConnection(
                        "jdbc:mysql://localhost:3306/likou?userSSL=false",
                        "root",
                        "root"
                );
                 statement = conn.createStatement();
            }

            /**
             * 具体业务逻辑实现的地方
             * 读取数据表中的数据并将其赋值给user对象
             * @param sourceContext
             * @throws Exception
             */
            @Override
            public void run(SourceContext<User> sourceContext) throws Exception {
                while (isRunning) {
                    //定义SQL查询语句
                    String sql="select id,username,password,name from user";
                    //执行SQL查询
                    ResultSet resultSet = statement.executeQuery(sql);
//                    将其封装到User中
                    while (resultSet.next()){
                        int id = resultSet.getInt("id");
                        String username = resultSet.getString("username");
                        String password = resultSet.getString("password");
                        String name = resultSet.getString("name");
                        sourceContext.collect(new User(
                                id,username,password,name
                        ));
                    }
                    Thread.sleep(10*1000L);
                    //每10s循环读取一次MySQL中的数据
                }
            }
            /**
             * 如果执行被取消，不再生成User对象
             */
            @Override
            public void cancel() {
                isRunning=false;
            }

            /**
             * 生命周期的结束，再次方法中定义收尾的操作
             * 关闭数据库连接，状态的连接，statement的连接
             * @throws Exception
             */
            @Override
            public void close() throws Exception {
                if(!statement.isClosed())statement.close();
                if(!conn.isClosed())conn.close();
            }
        }).printToErr();
        env.execute();

    }

    @Data
    @AllArgsConstructor
    public static class User{
        private int id;
        private String username;
        private String password;
        private String name;
    }

}

```

- **合流算子**
- **union**

```java
package sz.base.flink.transfromation;

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

/**
 * 使用union实现
 * 将以下数据进行取并集操作
 * 数据集1
 * "hadoop", "hive", "flume"
 * 数据集2
 * "hadoop", "hive", "spark"
 *
 * 注意:
 * 1: 合并后的数据不会自动去重
 * 2: 要求数据类型必须一致
 */
public class UnionDemo {
    public static void main(String[] args) throws Exception {
        /**
         * 实现步骤:
         * 1）初始化flink的流处理的运行环境
         * 2）加载/创建数据源
         * 3）处理数据
         * 4）打印输出
         * 5）递交执行作业
         */
        StreamExecutionEnvironment env = 
StreamExecutionEnvironment.getExecutionEnvironment();

        DataStreamSource<String> ds1 = env.fromElements("hadoop", "hive", "flume");
        DataStreamSource<String> ds2 = env.fromElements("hadoop","hive","spark");
        DataStream<String> result = ds1.union(ds2);
        result.printToErr();

        env.execute();
    }
}
```

- **connector**

```java
package sz.base.flink.transfromation;

import org.apache.flink.streaming.api.datastream.ConnectedStreams;
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.co.CoMapFunction;

/**
 *
 */
public class ConnectDemo {
    public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
        //生成两个数据流
        env.setParallelism(1);
        DataStreamSource<Integer> source = env.fromElements(1, 2, 3, 4, 5, 6, 7);
        DataStreamSource<String> source1 = env.fromElements("9", "10", "11", "12",
"13");
        ConnectedStreams<Integer, String> connect = source.connect(source1);
        SingleOutputStreamOperator<String> map = connect.map(new CoMapFunction<Integer,
String, String>() {
            @Override
            public String map1(Integer integer) throws Exception {
                return integer.toString();
            }

            @Override
            public String map2(String s) throws Exception {
                return s;
            }
        });
        map.printToErr();
        env.execute();
    }
}

```

- **分流**

```java
package sz.base.flink.transfromation;

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

/**
 * 将1~100之间的数据进行一个根据奇数和偶数进行分流操作
 * 侧输出流：单独的输出管道
 */
public class SplitStreamDemo {
    public static void main(String[] args) throws Exception {
        //1.创建流执行环境
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
        //2.设置并行度
        env.setParallelism(1);
        //3.生成数据源1-100序列
        DataStreamSource<Long> source = env.fromSequence(1, 100);
        //定义侧输出流保存偶数和奇数
        OutputTag<Long> odd = new OutputTag<>("odd", Types.LONG);
        OutputTag<Long> even = new OutputTag<>("even", Types.LONG);
        //4.1将偶数放到一个侧输出流中
        //4.2将奇数放到一个侧输出流中
        SingleOutputStreamOperator<Long> process = source.process(new
ProcessFunction<Long, Long>() {
            @Override
            public void processElement(Long aLong, Context context, Collector<Long>
collector) throws Exception {
                if (aLong % 2 == 0) {
                    context.output(even, aLong);
                } else {
                    context.output(odd, aLong);
                }
            }
        });
        //5.打印偶数或技术
        process.print("主干中的值");
        process.getSideOutput(odd).print("奇数");
        process.getSideOutput(even).print("偶数");
        //6.执行流环境
        env.execute("拆分数据流");

    }
}

```

- **物理分区**
- **5.重分区**

```java
package sz.base.flink.transfromation;

import org.apache.flink.api.common.functions.FilterFunction;
import org.apache.flink.api.common.functions.RichMapFunction;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.streaming.api.datastream.SingleOutputStreamOperator;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;

/**
 * 随机生成100个数字，过滤出来90个数字，封装taskid和数字，统计每个taskid
 * 最终的数字的个数是否都均匀，没有rebalance重分布，每个分区taskid对应的数字个数不均匀（倾斜）
 * 使用rebalance之后各个数字均匀
 * 开发步骤：
 * 0.获取流执行环境并设置并行度为3
 * 1.过滤出来90和数字[通过rebalance进行均衡]
 * 2.得到一个转换[taskid,1]
 * 3.taskid进行分组操作
 * 4.求和 - 每个cpu index处理的数字个数
 * 5.打印输出每个cpu index处理的个数
 * 6.执行流环境
 */
public class ReblalanceDemo {
    public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
        // * 0.获取流执行环境并设置并行度为3
        //90条数据，分给3个线程去处理，每个线程处理30条数据
        env.setParallelism(3);
        // * 1.过滤出来90和数字[通过rebalance进行均衡]
        SingleOutputStreamOperator<Long> source = env.fromSequence(1, 100).filter(new
FilterFunction<Long>() {
            @Override
            public boolean filter(Long aLong) throws Exception {
                return aLong > 10;
            }
        });
        // * 2.得到一个转换[taskid,1],,rebalance把数据重分布，平均分给每个cpu
        source.rebalance().map(new RichMapFunction<Long, Tuple2<Integer,Integer>>() {
            @Override
            public Tuple2<Integer, Integer> map(Long aLong) throws Exception {
                //获取上下文对象，在获取子任务的id
                int taskid = getRuntimeContext().getIndexOfThisSubtask();
                return Tuple2.of(taskid,1);
            }
        })
        // * 3.taskid进行分组操作
        .keyBy(t->t.f0)
        // * 4.求和 - 每个cpu index处理的数字个数
        .sum(1)
        // * 5.打印输出每个cpu index处理的个数
        .printToErr();
        // * 6.执行流环境
        env.execute();
    }

}

```

7.自定义分区

```java
package sz.base.flink.transfromation;

import org.apache.flink.api.common.functions.Partitioner;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.datastream.DataStreamSource;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;

public class CustomPartition {
    public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(3);
        DataStreamSource<String> source = env.fromElements("flink", "sqark", "123", "flink", "flink", "flink", "flink", "flink", "flink","hadoop");
        //自定义分区，将flink单独放到一个分区中，hadoop单独放到另一个分区中，其他放到另一个分区中
        DataStream<String> dataStream = source.partitionCustom(new MyPartition(), key -> key);
        dataStream.print();
        env.execute();
    }
    public static class MyPartition implements Partitioner<String>{
        @Override
        public int partition(String s, int i) {
            if(s.equals("flink")){
                return 0;
            }else if(s.equals("hadoop")){
                return 1;
            }else {
                return 2;
            }
        }
    }
}

```

DataSink数据输出

- **2.file和csv -- 方法弃用**

```java
package sz.base.flink.sink;

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


/**
 * 写出数据到文件系统
 */
public class SinkFileDemo {
    public static void main(String[] args) throws Exception {
        //1.创建流执行环境
        StreamExecutionEnvironment env = 
StreamExecutionEnvironment.getExecutionEnvironment();
        //2.设置并行度等参数
        env.setParallelism(1);
        //3.生成数据源
        DataStreamSource<String> node1 = env.socketTextStream("node1", 9999);
        //4.将数据写入到文件系统中,重复写入,并行写入文件形成一个文件夹放入数据
        node1.writeAsText(
                "data/output",
                FileSystem.WriteMode.OVERWRITE
        ).setParallelism(2);
        //写出格式为csv，指定行分隔符和列分隔符
/*        node1.writeAsCsv(
                "data/output",
                FileSystem.WriteMode.OVERWRITE,
                "\n","\001"
        );*/
        env.execute();
    }
}

```

connector连接器

MySQL写出

```java
package sz.base.flink.connector;

import org.apache.flink.api.common.typeinfo.Types;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.connector.jdbc.*;
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.util.Collector;

import java.sql.PreparedStatement;
import java.sql.SQLException;


/**
 * 通过connector连接器，将单词统计的值保存到MySQL数据表中
 * 开发步骤：
 * 1.导入包connector连接到jdbc的jar包
 * 2.创建数据库，并创建表
 * 3.创建流执行环境
 * 4.获取socket数据源
 * 5.将获取到文本映射成【word，1】
 * 6.keyBy分组和sum统计
 * 7.将结果.addSink（使用connector连接器）
 * 8.执行流环境
 * 9.查看结果
 */
public class SinkMySQLDemo {
    public static void main(String[] args) throws Exception {
        // * 3.创建流执行环境
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
        //设置并行度
        env.setParallelism(1);
        // * 4.获取socket数据源
        DataStreamSource<String> node1 = env.socketTextStream("node1", 9999);
        // * 2.创建数据库，并创建表
        // * 1.导入包connector连接到jdbc的jar包
        SingleOutputStreamOperator<Tuple2<String, Integer>> sum = node1.flatMap((String word, Collector<Tuple2<String, Integer>> out) -> {
            String[] words = word.split(" ");
            for (String s : words) {
                // * 5.将获取到文本映射成【word，1】
                out.collect(Tuple2.of(s, 1));
            }
            // * 6.keyBy分组和sum统计
        }).returns(Types.TUPLE(Types.STRING, Types.INT)).keyBy(t -> t.f0).sum(1);
        // * 7.将结果.addSink（使用connector连接器）
        sum.addSink(JdbcSink.sink(
                "insert into t_wordcount(word,counts) values (?,?) on duplicate key update counts=?",
                new JdbcStatementBuilder<Tuple2<String, Integer>>() {
                    @Override
                    public void accept(PreparedStatement ps, Tuple2<String, Integer> str) throws SQLException {
                        ps.setString(1, str.f0);
                        ps.setInt(2, str.f1);
                        ps.setInt(3, str.f1);
                    }
                },
                new JdbcExecutionOptions.Builder()
                        //5s自动将数据插入到数据表中
                        .withBatchIntervalMs(5000)
                        //每个批次插入的数量
                        .withBatchSize(1)
                        //最大重试的次数
                        .withMaxRetries(3)
                        .build(),
                new JdbcConnectionOptions.JdbcConnectionOptionsBuilder()
                        .withDriverName("com.mysql.jdbc.Driver")
                        .withUrl("jdbc:mysql://localhost:3306/likou?useSSL=false")
                        .withUsername("root")
                        .withPassword("root")
                        .build()
        ));
        // * 8.执行流环境
        env.execute();
        // * 9.查看结果
    }
}

```

kafka

消费数据

```java
package sz.base.flink.connector;

import org.apache.flink.api.common.serialization.SimpleStringSchema;
import org.apache.flink.streaming.api.datastream.DataStreamSource;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaConsumer;
import org.apache.kafka.clients.consumer.ConsumerConfig;

import java.util.Properties;

import static
org.apache.flink.streaming.connectors.kafka.FlinkKafkaConsumerBase.KEY_PARTITION_DISCOVERY_INTERVAL_MILLIS;

/**
 * 读取kafka集群中的数据通过connector
 * 开发步骤:
 * 1.获取流执行环境
 * 2.设置并行度
 * 3.Flink消费kafka的数据参数
 * 3.1配置参数
 * 3.2实例化FlinkKafkaConsumer 对象
 * 4.设置消费kafka的重启位置，如果当前程序崩溃了，从哪里接着消费
 * 5.将offset提交给flink的checkpoint来管理
 * 6.将生成的FlinkKafkaConsumer对象添加到source中
 * 7.打印输出数据
 * 8.执行流环境
 */
public class FlinKafkaReader {
    public static void main(String[] args) throws Exception {
        // * 1.获取流执行环境
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
        // * 2.设置并行度
        env.setParallelism(1);
        //开启checkpoint检查点
        env.enableCheckpointing(1000);
        // * 3.Flink消费kafka的数据参数
        // * 3.1配置参数
        Properties properties = new Properties();
        //kafka主机和端口(固定的)
        properties.setProperty(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
"node1:9092,node2:9092,node3:9092");
        //kafka groupID
        properties.setProperty(ConsumerConfig.GROUP_ID_CONFIG, "__consumer_src_");
        //kafka autocommit 将offset是否自动提交到kafka consumer 中保存
        properties.setProperty(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, "true");
        //自动分区发现
        properties.setProperty(KEY_PARTITION_DISCOVERY_INTERVAL_MILLIS, 6 * 1000 + "");
        // * 3.2实例化FlinkKafkaConsumer 对象
        FlinkKafkaConsumer<String> srctopic = new FlinkKafkaConsumer<>(
                "srctopic", new SimpleStringSchema(),
                properties
        );
        // * 4.设置消费kafka的重启位置，如果当前程序崩溃了，从哪里接着消费
        //从头
        srctopic.setStartFromEarliest();
        // * 5.将offset提交给flink的checkpoint来管理
        srctopic.setCommitOffsetsOnCheckpoints(true);
        // * 6.将生成的FlinkKafkaConsumer对象添加到source中
        DataStreamSource<String> source = env.addSource(srctopic);
        // * 7.打印输出数据
        source.printToErr();
        // * 8.执行流环境
        env.execute();
    }
}

```

生产数据到kafka

```java
package sz.base.flink.connector;

import org.apache.flink.api.common.serialization.SimpleStringSchema;
import org.apache.flink.streaming.api.datastream.DataStreamSource;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaConsumer;
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaProducer;
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.producer.ProducerConfig;

import java.util.Properties;

import static
org.apache.flink.streaming.connectors.kafka.FlinkKafkaConsumerBase.KEY_PARTITION_DISCOVERY_INTERVAL_MILLIS;

/**
 * Flink向kafka中写入数据
 * 开发步骤:
 * 1.获取流执行环境
 * 2.设置并行度
 * 3.Flink消费kafka的数据参数
 * 3.1配置参数
 * 3.2实例化FlinkKafkaConsumer 对象
 * 4.设置消费kafka的重启位置，如果当前程序崩溃了，从哪里接着消费
 * 5.将offset提交给flink的checkpoint来管理
 * 6.将生成的FlinkKafkaConsumer对象添加到source中
 * 7.打印输出数据
 * 8.执行流环境
 */
public class FlinKafkaWriter {
    public static void main(String[] args) throws Exception {
        // * 1.获取流执行环境
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
        // * 2.设置并行度
        env.setParallelism(1);
        //开启checkpoint检查点
        env.enableCheckpointing(1000);
        //从socket 数据源生成数据
        //将数据直接写入到kafka
        DataStreamSource<String> node1 = env.socketTextStream("node1", 9999);
        // * 3.Flink消费kafka的数据参数
        // * 3.1配置参数
        Properties properties = new Properties();
        //kafka主机和端口(固定的)
        properties.setProperty(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
"node1:9092,node2:9092,node3:9092");
        // * 3.2实例化FlinkKafkaConsumer 对象
        FlinkKafkaProducer<String> srctopic = new FlinkKafkaProducer<>(
                "srctopic", new SimpleStringSchema(),properties
                );
        node1.addSink(srctopic);
        // * 7.打印输出数据
        node1.printToErr();
        // * 8.执行流环境
        env.execute();
    }
}

```

window窗口

滑动窗口和滚动窗口

```java
package sz.base.flink.window;

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.functions.source.SourceFunction;
import org.apache.flink.streaming.api.windowing.assigners.SlidingEventTimeWindows;
import org.apache.flink.streaming.api.windowing.assigners.SlidingProcessingTimeWindows;
import org.apache.flink.streaming.api.windowing.assigners.TumblingProcessingTimeWindows;
import org.apache.flink.streaming.api.windowing.time.Time;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.Random;

/**
 * 统计全窗口和分流窗口，窗口大小是5s，计算单词的对应的数字之和
 * 输入的数据[apache,10][hadoop,2][flink,3][spark,5][hadoop,2]
 * 输出的数据
 * 窗口大小：5s window
 * 输出数据 keyBy,聚合：[hadoop,10][apache,20][spark,25]...
 * 全窗口,不分流直接求和[hadoop,55]
 * 开发步骤：
 * 1.获取流执行环境
 * 2.获取随机数据源 GenerateRandomNumEverySecond
 * 3.全窗口滚动处理时间窗口为5s ，统计sum求和
 * 4.根据单词进行分区，滚动处理时间窗口为5s ，统计sum求和
 * 5.打印输出
 * 6.执行流环境
 * <p>
 * window api  格式
 * 数据流.keyBy(分组字段) -- 分流操作
 * .window(窗口的类型)   --时间窗口（滚动时间、滑动时间、会话时间）计数窗口
 * .windowAll(窗口的类型) -- 窗口     没有用keyBy用windowAll,用了keyBy用window
 * .trigger(触发的时间)     -- 触发,默认触发方式
 * .allowedLateness(允许延迟的时间)  --3min 在3min中之内来的数据依然会被计算
 * .sideOutputLateData(侧输出流) output tag -->超过3min，将这些数据流保存的位置  Side OutputTag 可以取出来
 * .聚合函数() reduce / aggregate / fold / apply() / process()全量主要 -- 聚合函数 ①增量(来一条处理一条) ②全量 (窗口内数据聚合)
 */
public class TimeWindowDemo01_1 {
    public static void main(String[] args) throws Exception {
        //1.获取流执行环境
        StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
        //2.设置并行度
        env.setParallelism(1);
        //3.获取数据源
        DataStreamSource<Tuple2<String, Integer>> source = env.addSource(new GenerateRandomNumEverySecond());
        //4.窗口的划分和聚合计算
        //4.1windowall 全窗口，5s 内所有的元素聚合，不分类keyBy
        /*source.windowAll(TumblingProcessingTimeWindows.of(Time.seconds(5))) //会统计每5s的value累加，不管key是什么会输出第一个进来的key值
                .sum(1).printToErr();*/
        //4.2keyBy ... window 先分组再开窗
        source.keyBy(k->k.f0)
        //5.对开窗的数据求和,窗口分类：①时间窗口：滚动时间、滑动时间、会话时间  ② 技术窗口：滚动计数、滑动计数窗口
                //窗口API：  窗口名称 + 时间
                /*.window(TumblingProcessingTimeWindows.of(Time.seconds(5))).sum(1).printToErr();*/
        //偏移量,用来偏移(Time.days(1), Time.hours(-8))时间，一般是中国时间如左例
        //.window(TumblingProcessingTimeWindows.of(Time.seconds(5),Time.seconds(2))).sum(1).printToErr();
                //计算,每2s（滑动时间）计算5s（窗口）
                .window(SlidingProcessingTimeWindows.of(Time.seconds(5),Time.milliseconds(2100))).sum(1).printToErr();
        //偏移量,用来偏移Time.hours(12), Time.hours(1), Time.hours(-8))时间，一般是中国时间如左例
                //.window(SlidingProcessingTimeWindows.of(Time.seconds(5),Time.seconds(2)),Time.seconds(1)).sum(1).printToErr();
        //6.打印输出结果
        //7.执行流环境
        env.execute();
    }

    /**
     * 实现一个SourceFunction，每一秒好创建一个Tuple2
     */
    public static class GenerateRandomNumEverySecond implements SourceFunction<Tuple2<String, Integer>> {
        boolean isRunning = true;
        //随机数
        final Random rm = new Random();
        //定义一个数组/集合
        List<String> keys = Arrays.asList("hadoop", "spark", "flink", "hadoop", "hive");

        /**
         * 核心业务逻辑，每秒生成二元元组[hadoop/spark,随机数字]
         *
         * @param sourceContext
         * @throws Exception
         */
        @Override
        public void run(SourceContext<Tuple2<String, Integer>> sourceContext) throws Exception {

            while (isRunning) {
                //获取列表中的随机的key值
                String key = keys.get(rm.nextInt(keys.size()));
                //获取一个value= [0 ~ 50]之间的值
                int value = rm.nextInt(10);
                //返回Tuple2
                Tuple2<String, Integer> of = Tuple2.of("spark", value);
                sourceContext.collect(of);
                System.out.println(of);
                //要求每1s打印一条数据
                Thread.sleep(1000);

            }
        }

        @Override
        public void cancel() {
            isRunning = false;
        }
    }
}

```

会话窗口

```
package sz.base.flink.window;

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.functions.source.SourceFunction;
```

```java
import org.apache.flink.streaming.api.windowing.assigners.ProcessingTimeSessionWindows;
import org.apache.flink.streaming.api.windowing.time.Time;

import java.text.SimpleDateFormat;
import java.util.Arrays;
import java.util.Date;
import java.util.List;
import java.util.Random;

/**
 * 会话窗口
 * 会话窗口 -
 * 需求1：定义一个会话时间窗口，5sgap，统计全量windowall（Non-key）数据之和
 * 需求2：定义一个会话时间窗口，5s gap ，统计按照key分组后的每个组数据内的数字和
 */
public class SessionWindowDemo02_2 {
    public static void main(String[] args) throws Exception {
        //获取流执行环境
        StreamExecutionEnvironment env = 
StreamExecutionEnvironment.getExecutionEnvironment();
        //设置并行度
        env.setParallelism(1);
        //添加自定义数据源
        DataStreamSource<Tuple2<String, Integer>> source = env.addSource(new 
GenerateRandomNumRandomSecond());
        //1.全窗口windowALl，non-keyed window 设置处理时间session窗口，间隔5s求和

/*source.windowAll(ProcessingTimeSessionWindows.withGap(Time.seconds(5))).sum(1).printTo
Err();*/
        //2.根据单词设置处理时间session窗口，间隔5s，求和
        source.keyBy(k-
>k.f0).window(ProcessingTimeSessionWindows.withGap(Time.seconds(5))).sum(1).printToErr()
;
        //执行
        env.execute();
    }

    /*
   自定义Source
   每隔随机时间(1~7秒之间)产生一个的k,v  k是hadoop spark flink 其中某一个，v是随机数字
    */
    public static class GenerateRandomNumRandomSecond implements SourceFunction<Tuple2<String, Integer>> {
        private SimpleDateFormat sdf =new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        private boolean isRun = true;
        private final Random random = new Random();
        private final List<String> keyList = Arrays.asList("hadoop", "spark", "flink");
        @Override
        public void run(SourceContext<Tuple2<String, Integer>> ctx) throws Exception {
            while (this.isRun) {
                String key = keyList.get(random.nextInt(3));
                Tuple2<String, Integer> value = Tuple2.of(key, random.nextInt(9));
                ctx.collect(value);
                long sleepTime = 5000L;
                while (sleepTime == 5000L) {
                    sleepTime = random.nextInt(7) * 1000L;
                }
                System.out.println(sdf.format(new Date()) + ":---will sleep " + sleepTime + " ms---: " + value);
                Thread.sleep(sleepTime);
            }
        }

        @Override
        public void cancel() {
            this.isRun = false;
        }
    }
}

```

计数窗口

```java
package sz.base.flink.window;

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.functions.source.SourceFunction;

import java.text.SimpleDateFormat;
import java.util.Arrays;
import java.util.Date;
import java.util.List;
import java.util.Random;

/**
 * 随机生成数据[hadoop/flink,随机生成数字20/30], 使用计数窗口进行计算
 * 需求1: 通过滚动计数 window all non-keyed
 * 需求2: 通过滑动计数 keyBy window
 */
public class CountWindowDemo03_3 {
    public static void main(String[] args) throws Exception {
        //计算每5个计算non-keyed 窗口内的数据
        //1.创建流环境
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
        //设置参数
        env.setParallelism(1);
        //获取数据源
        DataStreamSource<Tuple2<String, Integer>> source = env.addSource(new
GenerateRandomNumRandomSecond());
        //窗口计算
        /*source.countWindowAll(5).sum(1).printToErr();*/
        //先分组在count window
        /*source.keyBy(t->t.f0).countWindow(5).sum(1).printToErr();*/
        //先分组在滑动 count window 每2条计算前5条 如果分组则统计同一组内
        source.keyBy(t->t.f0).countWindow(5,2).sum(1).printToErr();
        env.execute();
    }
    /*
  自定义Source
  每隔随机时间(1~7秒之间)产生一个的k,v  k是hadoop spark flink 其中某一个, v是随机数字
   */
    public static class GenerateRandomNumRandomSecond implements
SourceFunction<Tuple2<String, Integer>> {
        private SimpleDateFormat sdf =new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        private boolean isRun = true;
        private final Random random = new Random();
        private final List<String> keyList = Arrays.asList("hadoop", "spark", "flink");
        @Override
        public void run(SourceContext<Tuple2<String, Integer>> ctx) throws Exception {
            while (this.isRun) {
                String key = keyList.get(random.nextInt(3));
                Tuple2<String, Integer> value = Tuple2.of(key, random.nextInt(9));
                ctx.collect(value);
                System.out.println("------: " + value);
                Thread.sleep(1000);
            }
        }

        @Override
        public void cancel() {
            this.isRun = false;
        }
    }
}

```

计数窗口2

```java
package sz.base.flink.window;

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.ProcessWindowFunction;
import org.apache.flink.streaming.api.windowing.windows.GlobalWindow;
import org.apache.flink.util.Collector;

import java.util.Arrays;

/**
 * 每5位同学统计一下这五位同学的平均成绩
 * 输入的数据["class1","姓名",100L]
 * 输出的数据:  每5个同学计算出平均分
 * .聚合函数() reduce / aggregate / fold / apply() / process()全量主要 -- 聚合函数 ①增量(来一条处理一条) ②全量 (窗口内数据聚合)
 */
public class WindowCountProcessDemo_4 {
    public static void main(String[] args) throws Exception {
        //获取流环境
        StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
        //设置并行度
        env.setParallelism(2);
        //获取数据源
        DataStreamSource<Tuple3<String, String, Long>> source = env.fromCollection(Arrays.asList(ENGLISH));
        //分班开窗，每5个同学分到一个组内，全量的计算，计算平均分
        SingleOutputStreamOperator<Double> process = source.shuffle().keyBy(k -> k.f0)
                .countWindow(5)
                //全窗口计算 方法传入的参数
                //T: 输入流的类型
                // R:  输出的类型
                //K :  分组的字段类型
                //W:  开窗的类型 ① 时间窗口类型 ② 计数窗口类型
                .process(new ProcessWindowFunction<Tuple3<String, String, Long>, Double, String, GlobalWindow>() {
                    /**
                     * 核心处理逻辑，5个同学的平均分
                     * @param s 分组的字段，班级
                     * @param context 上下文对象
                     * @param iterable  就是这个计数窗口内的所有数据
                     * @param collector
                     * @throws Exception
                     */
                    @Override
                    public void process(String s, Context context, Iterable<Tuple3<String, String, Long>> iterable, Collector<Double> collector) throws Exception {
                        //定义一个变量用于接收所有同学的成绩
                        Long sum = 0L;
                        for (Tuple3<String, String, Long> stu : iterable) {
                            sum += stu.f2;
                        }
                        //计算平均值
                        double result = sum / 5.00D;
                        //收集数据
                        collector.collect(result);
                    }
                });
        //得到窗口的结果
        //打印输出结果
        process.print();
        //执行流环境
        env.execute();
    }

    public static final Tuple3<String, String, Long>[] ENGLISH = new Tuple3[]{
            Tuple3.of("class1", "张三", 100L),
            Tuple3.of("class2", "小七", 59L),
            Tuple3.of("class1", "李四", 78L),
            Tuple3.of("class1", "小七", 59L),
            Tuple3.of("class2", "李四", 78L),
            Tuple3.of("class2", "王五", 99L),
            Tuple3.of("class1", "王五", 99L),
            Tuple3.of("class1", "赵六", 81L),
            Tuple3.of("class2", "赵六", 81L),
            Tuple3.of("class2", "张三", 100L),
    };
}

```

水印机制

水印机制

```java
package sz.base.flink.watermark;

import org.apache.flink.api.common.eventtime.WatermarkStrategy;
import org.apache.flink.api.common.functions.FlatMapFunction;
import org.apache.flink.api.common.functions.MapFunction;
import org.apache.flink.api.common.serialization.SimpleStringSchema;
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.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaConsumer;
import org.apache.flink.util.Collector;
import org.apache.kafka.clients.consumer.ConsumerConfig;

import java.time.Duration;
import java.util.Properties;

/**
 * 实现数据源上的水印机制，source端设置水印机制
 * 开发步骤:
 * 1.获取流环境
 * 2.读取kafka source ， 初始化了FlinkKafkaConsumer
 * 3.source 设置水印机制
 * 4.添加数据源
 * 5.wordcount
 * 6.打印输出
 * 7.执行流环境
 */
public class KafkaSourceWatemarkDemo_5 {
    public static void main(String[] args) throws Exception {
        //1.获取流环境
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
        //设置参数
        env.setParallelism(1);
        env.enableCheckpointing(2000);
        //2.读取kafka
        Properties properties = new Properties();

properties.setProperty(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,"node1:9092,node2:9092,node3:9092");
        properties.setProperty(ConsumerConfig.GROUP_ID_CONFIG,"__consumer_src_");
        FlinkKafkaConsumer<String> srctopic = new FlinkKafkaConsumer<>(
                "srctopic",
                new SimpleStringSchema(),
                properties
        );
        //分配提交checkpoint 维护offset
        srctopic.setCommitOffsetsOnCheckpoints(true);
        //如果失败了，从哪里继续来读取,从组内
        srctopic.setStartFromGroupOffsets();
        //3.设置水印
        //作用就是，在source端就将乱序的数据排个序
        srctopic.assignTimestampsAndWatermarks(
                //选择乱序时间，等待30s
                WatermarkStrategy.forBoundedOutOfOrderness(Duration.ofSeconds(30))
        );
        //4.添加数据源
        DataStreamSource<String> source = env.addSource(srctopic);
        //
        source.flatMap(new FlatMapFunction<String, String>() {
            @Override
            public void flatMap(String s, Collector<String> collector) throws Exception
{
                String[] words = s.split(" ");
                for (String word : words) {
                    collector.collect(word);
                }
            }
        }).map(new MapFunction<String, Tuple2<String, Integer>>() {
            @Override
            public Tuple2<String, Integer> map(String s) throws Exception {
                return Tuple2.of(s, 1);
            }
        }).keyBy(new KeySelector<Tuple2<String, Integer>, String>() {
            @Override
            public String getKey(Tuple2<String, Integer> stringIntegerTuple2) throws
Exception {
                return stringIntegerTuple2.f0;
            }
        }).sum(1).print();
        //执行流环境
        env.execute();
    }
}

```

assignTimestampsAndWatermarks

forBoundedOutOfOrderness

- 分配水印机制，单调乱序的水印，会有最大的延迟时间 (最新用法)，乱序实现

```
package sz.base.flink.watermark;
```

```java

import lombok.AllArgsConstructor;
import lombok.Data;
import org.apache.commons.collections.IteratorUtils;
import org.apache.flink.api.common.eventtime.WatermarkStrategy;
import org.apache.flink.api.common.functions.MapFunction;
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.ProcessWindowFunction;
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;

/**
 * 在单调递增的水印，在非source上添加水印
 *1.定义类 WaterSensor  String id; Long ts; Integer vc;
 * 2.创建流执行环境
 * 3.获取socket文本数据
 * 4.将字符串数据切分成 WaterSensor 对象数据
 * 5.分配水印机制，单调递增
 * 6.分配后的数据根据id进行分组
 * 7.设置滚动事件时间窗口，时间为10秒
 * 8.对开窗数据进行process
 */
public class WaterSensorDemo_6 {
    public static void main(String[] args) throws Exception {
        //1.定义类 WaterSensor  String id; Long ts; Integer vc;
        //2.创建流执行环境
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(1);
        //3.获取socket文本数据
        DataStreamSource<String> node1 = env.socketTextStream("node1", 9999);
        //4.将字符串数据切分成 WaterSensor 对象数据
        SingleOutputStreamOperator<WaterSensor> operator = node1.map(new
MapFunction<String, WaterSensor>() {
```

```
            @Override
            public WaterSensor map(String s) throws Exception {
                String[] split = s.split(",");
                return new WaterSensor(
                        split[0],
                        Long.parseLong(split[1]),
                        Integer.parseInt(split[2])
                );
            }
        });
        //5.分配水印机制，单调递增,默认为0s 分配水印机制，单调乱序的水印，会有最大的延迟时间(最新用法)
        SingleOutputStreamOperator<WaterSensor> waterSensorSingleOutputStreamOperator = operator.assignTimestampsAndWatermarks(WatermarkStrategy.<WaterSensor>forBoundedOutOfOrderness(Duration.ofSeconds(3)).withTimestampAssigner((element, recordTimestamp)->element.getTs()*1000));
        //6.分配后的数据根据id进行分组
        SingleOutputStreamOperator<String> process = waterSensorSingleOutputStreamOperator.keyBy(waterSensor -> waterSensor.id)
                //7.设置滚动事件时间窗口，时间为10秒
                .window(TumblingEventTimeWindows.of(Time.seconds(10)))
                //8.对开窗数据进行process
                .process(new ProcessWindowFunction<WaterSensor, String, String, TimeWindow>() {
                    @Override
                    public void process(String id, Context context, Iterable<WaterSensor> iterable, Collector<String> collector) throws Exception {
                        String str = "id=" + id + "\n" + "数据应为:  " + iterable + "\n" + "数据条数:  " + IteratorUtils.toList(iterable.iterator()).size() + "\n" + "窗口开始时间:  "
                                + context.window().getStart() + ",窗口结束时间:  " + context.window().getEnd() + "\n" + "----------------------";
                        collector.collect(str);
                    }
                });
        //9.打印输出
        process.print();
        //10.执行流环境
        env.execute();
    }

    @Data
    @AllArgsConstructor
```

```java
    public static class WaterSensor{
        private String id;
        private Long ts;
        private Integer vc;

    }
}

```

forMonotonousTimestamps

- 单调递增,默认为0s

```java
package sz.base.flink.watermark;

import lombok.AllArgsConstructor;
import lombok.Data;
import org.apache.commons.collections.IteratorUtils;
import org.apache.flink.api.common.eventtime.WatermarkStrategy;
import org.apache.flink.api.common.functions.MapFunction;
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.ProcessWindowFunction;
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;

public class WaterSensorOutofOrdemessDemo_7 {
    public static void main(String[] args) throws Exception {
        //1.定义类 WaterSensor  String id; Long ts; Integer vc;
        //2.创建流执行环境
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(1);
        //3.获取socket文本数据
        DataStreamSource<String> node1 = env.socketTextStream("node1", 9999);
        //4.将字符串数据切分成 WaterSensor 对象数据
        SingleOutputStreamOperator<WaterSensor> operator = node1.map(new MapFunction<String, WaterSensor>() {
            @Override
            public WaterSensor map(String s) throws Exception {
                String[] split = s.split(",");
                return new WaterSensor(
                        split[0],
                        Long.parseLong(split[1]),
                        Integer.parseInt(split[2])
                );
            }
        });
        //5.分配水印机制，单调递增,默认为0s
        SingleOutputStreamOperator<WaterSensor> waterSensorSingleOutputStreamOperator = operator.assignTimestampsAndWatermarks(WatermarkStrategy.<WaterSensor>forMonotonousTimestamps().withTimestampAssigner((element, recordTimestamp) -> element.getTs() * 1000));
        //6.分配后的数据根据id进行分组
        SingleOutputStreamOperator<String> process = waterSensorSingleOutputStreamOperator.keyBy(waterSensor -> waterSensor.id)
                //7.设置滚动事件时间窗口，时间为10秒
                .window(TumblingEventTimeWindows.of(Time.seconds(10)))
                //8.对开窗数据进行process
                .process(new ProcessWindowFunction<WaterSensor, String, String, TimeWindow>() {
                    @Override
                    public void process(String id, Context context, Iterable<WaterSensor> iterable, Collector<String> collector) throws Exception {
                        String str = "id=" + id + "\n" + "数据应为：  " + iterable + "\n" + "数据条数：  " + IteratorUtils.toList(iterable.iterator()).size() + "\n" + "窗口开始时间：  "
                                + context.window().getStart() + ",窗口结束时间：  " + context.window().getEnd() + "\n" + "----------------------";
                        collector.collect(str);
                    }
                });
        //9.打印输出
        process.print();
        //10.执行流环境
        env.execute();
    }

    @Data
    @AllArgsConstructor
    public static class WaterSensor {
        private String id;
        private Long ts;
        private Integer vc;

    }
}


```

允许迟到时间allowedLateness

```java
package sz.base.flink.allowlateness;

import org.apache.commons.collections.IteratorUtils;
import org.apache.flink.api.common.eventtime.WatermarkStrategy;
import org.apache.flink.api.common.functions.MapFunction;
import org.apache.flink.api.java.tuple.Tuple2;
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.ProcessWindowFunction;
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;

/**
 * 需求- 根据socket输入的个数
 * 来统计一下在指定时间窗口内一共有多少个元素
 * 输入的格式：hello,1
 * 输出的格式：根据时间窗口，得到一个个数 10 ，窗口的开始时间和结束时间
 */
public class AllowLatenssDemo_8 {
    public static void main(String[] args) throws Exception {
        //获取流执行环境
        StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(1);
        DataStreamSource<String> node1 = env.socketTextStream("node1", 9999);
        SingleOutputStreamOperator<Tuple2<String, Integer>> operator = node1.map(new MapFunction<String, Tuple2<String, Integer>>() {
            @Override
            public Tuple2<String, Integer> map(String s) throws Exception {
                String[] lines = s.split(",");
                return Tuple2.of(lines[0], Integer.parseInt(lines[1]));
            }
        });
        //分配水位线，最大延迟3s
        SingleOutputStreamOperator<String> process = operator.assignTimestampsAndWatermarks(WatermarkStrategy.<Tuple2<String, Integer>>forBoundedOutOfOrderness(Duration.ofSeconds(3)).withTimestampAssigner((element, recordTimestamp) -> element.f1 * 1000))
                .keyBy(t -> t.f0)
                .window(TumblingEventTimeWindows.of(Time.seconds(5)))
                //允许最大严重乱序时间为2s,触发计算窗口之后的延迟
                .allowedLateness(Time.seconds(2))
                //process处理，对窗口数据中的元素进行统计，生成[单词,出现次数]，并将窗口开始时间和结束时间打印到控制台
                .process(new ProcessWindowFunction<Tuple2<String, Integer>, String, String, TimeWindow>() {
                    @Override
                    public void process(String s, Context context, Iterable<Tuple2<String, Integer>> iterable, Collector<String> collector) throws Exception {
                        int size = IteratorUtils.toList(iterable.iterator()).size();
                        collector.collect(s + ":" + size);
                        System.out.println("当前窗口的start: " + context.window().getStart() + " 窗口结束时间: " + context.window().getEnd());
                    }


                });
        process.print();
        env.execute();
    }
}

```

侧输出流sideallowedLateness

```java
package sz.base.flink.allowlateness;

import org.apache.commons.collections.IteratorUtils;
import org.apache.flink.api.common.eventtime.WatermarkStrategy;
import org.apache.flink.api.common.functions.MapFunction;
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.datastream.SingleOutputStreamOperator;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.functions.windowing.ProcessWindowFunction;
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;

/**
 * 需求- 根据socket输入的个数
 * 来统计一下在指定时间窗口内一共有多少个元素
 * 输入的格式：hello,1
 * 输出的格式：根据时间窗口，得到一个个数 10 ， 窗口的开始时间和结束时间
 */
public class AllowLatenssSideDemo_9 {
    public static void main(String[] args) throws Exception {
        //获取流执行环境
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(1);
        DataStreamSource<String> node1 = env.socketTextStream("node1", 9999);
        SingleOutputStreamOperator<Tuple2<String, Integer>> operator = node1.map(new
MapFunction<String, Tuple2<String, Integer>>() {
            @Override
            public Tuple2<String, Integer> map(String s) throws Exception {
                String[] lines = s.split(",");
        return Tuple2.of(lines[0], Integer.parseInt(lines[1]));
      }
    });
    OutputTag<Tuple2<String, Integer>> seriousLade = new OutputTag<>(
        "seriousLadeDate",
        Types.TUPLE(Types.STRING, Types.INT)
    );
    //分配水位线，最大延迟3s
    SingleOutputStreamOperator<String> process = operator.assignTimestampsAndWatermarks(WatermarkStrategy.<Tuple2<String, Integer>>forBoundedOutOfOrderness(Duration.ofSeconds(3)).withTimestampAssigner((element, recordTimestamp) -> element.f1 * 1000))
        .keyBy(t -> t.f0)
        .window(TumblingEventTimeWindows.of(Time.seconds(5)))
        //允许最大严重乱序时间为2s,触发计算窗口之后的延迟
        .allowedLateness(Time.seconds(2))
        //超过最大严重乱序时间保存到哪里，保存到侧输出流
        .sideOutputLateData(seriousLade)
        //process处理，对窗口数据中的元素进行统计，生成[单词,出现次数]，并将窗口开始时间和结束时间打印到控制台
        .process(new ProcessWindowFunction<Tuple2<String, Integer>, String, String, TimeWindow>() {
          @Override
          public void process(String s, Context context, Iterable<Tuple2<String, Integer>> iterable, Collector<String> collector) throws Exception {
            int size = IteratorUtils.toList(iterable.iterator()).size();
            collector.collect(s + ":" + size);
            System.out.println("当前窗口的start: " + context.window().getStart() + " 窗口结束时间: " + context.window().getEnd());
          }

        });
    process.print();
    process.getSideOutput(seriousLade).print();
    env.execute();
  }
}

```

state状态

```java
package sz.base.flink.keyedstate;

import org.apache.flink.api.common.functions.FlatMapFunction;
import org.apache.flink.api.common.functions.RichReduceFunction;
import org.apache.flink.api.common.state.ValueState;
import org.apache.flink.api.common.state.ValueStateDescriptor;
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.datastream.KeyedStream;
import org.apache.flink.streaming.api.datastream.SingleOutputStreamOperator;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.util.Collector;

/**
 * 单词统计
 * 主要用于flink自带的state管理的算子 sum, 本身是支持state状态
 * 自定义一个 state ->  ValueState 保存每次聚合的中间结果 来实现单词次数的聚合
 * state的数据结构分类:
 * 1.ValueState:  存储单值
 * 2.ListState:  存储的是值的集合
 * 3.MapState: 存储的是key-value 键值对
 * 4.BroadcastState :  广播状态, 使用在 broadcast (广播状态), 用于广播变量中
 * 5.ReduceState:  聚合状态
 * <p>
 * 分析:
 * 1.做一个常规的wordcount, sum去看单词统计的结果
 * 2.自定义适用ValueState 保存每次聚合的中间结果 来实现单词次数的聚合
 * <p>
 * 开发步骤:
 * 1.创建流环境
 * 2.设置并行度参数
 * 3.构建socket 数据源
 * 4.每行输入数据的单词的拆分
 * 5.根据 word 进行分组/分流
 * 6.先用 sum 进行求和 (自带的算子带状态)
 * 7.使用 reduce 聚合 (没有状态) 的算子, 如何去做
 * 8.打印输出的结果
 * 9.执行流环境
 */


public class WordcountKeyedState_1 {
    public static void main(String[] args) throws Exception {
        // * 1.创建流环境
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
        // * 2.设置并行度参数
        env.setParallelism(1);
        // * 3.构建socket 数据源
        DataStreamSource<String> node1 = env.socketTextStream("node1", 9999);
        // * 4.每行输入数据的单词的拆分
        SingleOutputStreamOperator<Tuple2<String, Integer>> flatMap = node1.flatMap(new
FlatMapFunction<String, Tuple2<String, Integer>>() {
            @Override
            public void flatMap(String s, Collector<Tuple2<String, Integer>> collector)
throws Exception {
                String[] lines = s.split(",");
                for (String line : lines) {
                    collector.collect(Tuple2.of(line, 1));
                }
            }
        });
        // * 5.根据 word 进行分组/分流
        KeyedStream<Tuple2<String, Integer>, String> keyedStream = flatMap.keyBy(t ->
t.f0);
        // * 6.先用 sum 进行求和（自带的算子带状态）
//          SingleOutputStreamOperator<Tuple2<String, Integer>> result =
keyedStream.sum(1);
        // * 7.使用 reduce 聚合（没有状态）的算子，如何去做
        SingleOutputStreamOperator<Tuple2<String, Integer>> result =
keyedStream.reduce(new RichReduceFunction<Tuple2<String, Integer>>() {
            //定义状态的描述器,中间结果状态
            ValueState<Tuple2<String, Integer>> reduceState = null;

            /**
             * 初始化工作
             * 获取ValueState，用于保存或读取中间结果state的值（中间结果值）
             * @param parameters
             * @throws Exception
             */
            @Override
            public void open(Configuration parameters) throws Exception {
                //从上下文变量获取
                reduceState = getRuntimeContext().getState(new
ValueStateDescriptor<Tuple2<String, Integer>>("reduceState", Types.TUPLE(Types.STRING,
Types.INT)));

            }

            /**
             * 做值的累加，获取ValueState中的值和当前的值进行累加，保存到状态中
             * @param stringIntegerTuple2
             * @param t1
             * @return
             * @throws Exception
             */
            @Override
            public Tuple2<String, Integer> reduce(Tuple2<String, Integer>
stringIntegerTuple2, Tuple2<String, Integer> t1) throws Exception {
                Tuple2<String, Integer> value = reduceState.value();
                //第一次存的时候，没有值，需要对状态赋值
                if (value == null) {
                    value = stringIntegerTuple2;
                }
                //更新中间结果
                reduceState.update(Tuple2.of(value.f0, value.f1 + t1.f1));
                return Tuple2.of(value.f0, value.f1 + t1.f1);
            }


            @Override
            public void close() throws Exception {
                System.out.println(reduceState.value().f0 + "--" +
reduceState.value().f1);
            }
        });
        // * 8.打印输出的结果
        result.print();
        // * 9.执行流环境
        env.execute();
    }
}

```

operator state

```
package sz.base.flink.ouperator;

import org.apache.flink.api.common.restartstrategy.RestartStrategies;
import org.apache.flink.api.common.state.ListState;
import org.apache.flink.api.common.state.ListStateDescriptor;
import org.apache.flink.api.common.typeinfo.Types;
import org.apache.flink.runtime.state.FunctionInitializationContext;
import org.apache.flink.runtime.state.FunctionSnapshotContext;
import org.apache.flink.streaming.api.checkpoint.CheckpointedFunction;
import org.apache.flink.streaming.api.datastream.DataStreamSource;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.functions.source.SourceFunction;

import java.awt.peer.CheckboxMenuItemPeer;

/**
 * 需求：模拟一个消费kafka，将读取每一条数据都记录一个offset+=1，将offset 偏移量保存到
 * operator state 中，每生成 5 条数据，模拟个bug， 让程序自动重启，接着上次保存的最新的offset 接着执行。
 * Operator state 实现接口：
 * 1.ListCheckpointed
 * 2.CheckpointedFunction
 * 开发步骤：
 */
public class OperatorStateDemo_2 {
    public static void main(String[] args) throws Exception {
        //1.创建流执行环境，设置并行度
        StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(1);
        //2.启用checkpoint ，每1s 做一次 checkpoint
        env.enableCheckpointing(1000);
        //3.设置重启策略，程序挂掉，自动重新启动起来
```

```java
        env.setRestartStrategy(RestartStrategies.fixedDelayRestart(3, 3000));
        //4.添加自定义数据源，将模拟的 offset 返回
        DataStreamSource<Long> source = env.addSource(new MySource());
        //5.map操作，将Long 类型转换成字符串类型并打印输出
        source.map(t->t.toString()).print();
        //6.执行流环境
        env.execute();
    }

    private static class MySource implements SourceFunction<Long>, CheckpointedFunction {
        //自定义source 实现 SourceFunction<Integer>和 CheckpointedFunction
        //1.定义变量
        //1.1定义标记用于循环生成标记
        boolean isRunning = true;
        //1.2定义 currentCounter 用于保存当前的计数值
        Long currentCounter = 0L;
        //1.3定义LIstState
        ListState<Long> offsetState = null;
        //1.4定义ListState 描述


        //2.重写SnapshotState 方法，清除状态和将最新的累加值添加到状态中 (给当前状态做一个快照)
        @Override
        public void snapshotState(FunctionSnapshotContext functionSnapshotContext) throws Exception {
            //将上一次的 state 状态清空,然后最新的添加进去(checkpoint记录的(累加器)偏移量,中间状态),只保存一个最新的中间状态
            offsetState.clear();
            offsetState.add(currentCounter);
        }

        //3.重写 initializeState方法，获取状态并遍历Iterable，将其赋值给 currentCounter
        @Override
        public void initializeState(FunctionInitializationContext context) throws Exception {
            //3.1将历史存储(checkpoint中)的状态遍历出来 赋值给currentCounter变量
            offsetState = context.getOperatorStateStore().getListState(
                    new ListStateDescriptor<Long>(
                            "offsetState", Types.LONG
                    )

            );
            //将从 operator state 中读取最新 offset 赋值给 currentCounter
            Iterable<Long> longs = offsetState.get();
            //将 List 中最新的 offset 赋值给 currentCounter
            for (Long aLong : longs) {
                currentCounter = aLong;
            }
        }

        //4.重写run方法，每秒循环收集累加的counter，每5个生成一个异常
        @Override
        public void run(SourceContext<Long> sourceContext) throws Exception {
            //4.1 持续循环
            while (isRunning) {
                //4.2 对currentCounter 累加 读一条+1，相当于偏移量
                currentCounter++;
                //4.3 收集当前累加的值
                sourceContext.collect(currentCounter);
                //4.4休眠1s
                Thread.sleep(1000);
                //4.5 如果是5的倍数就模拟输出异常
                if (currentCounter % 5 == 0) {
                    throw new RuntimeException("出错了，出bug了");
                }
            }
        }

        //5.重写cancel方法
        @Override
        public void cancel() {
            isRunning = false;
        }
    }
}

```

状态有效期TTL

```java
package sz.base.flink.keyedstate;


import org.apache.commons.io.FileUtils;
import org.apache.flink.api.common.functions.FlatMapFunction;
import org.apache.flink.api.common.functions.RichFlatMapFunction;
import org.apache.flink.api.common.functions.RichReduceFunction;
import org.apache.flink.api.common.state.StateTtlConfig;
import org.apache.flink.api.common.state.ValueState;
import org.apache.flink.api.common.state.ValueStateDescriptor;
import org.apache.flink.api.common.time.Time;
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.datastream.KeyedStream;
import org.apache.flink.streaming.api.datastream.SingleOutputStreamOperator;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.functions.source.SourceFunction;
import org.apache.flink.util.Collector;

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

/**
 * 需求- 读取文件数据，每3s读取一行，然后将读取到的字符串按照空格拆分计算每个单词出现的次数
 * 使用到state ttl ，将TTL 设置为 6s ，查看统计的结果
 * 输入：
 * 文件中的所有的单词，每行1~n单词，每3s 读取一行
 * 输出：
 * hello,1
 * hello,2
 * spark,2
 * flink,3
 * hello,1
 */
public class WordcountTTLState_3 {
    public static void main(String[] args) throws Exception {
        // * 1.创建流环境
        StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
        // * 2.设置并行度参数
        env.setParallelism(1);
        // * 3.读取文件中的数据，每3s中读取一行
        DataStreamSource<String> node1 = env.addSource(new SourceFunction<String>() {
            boolean isRunning = true;

            @Override
            public void run(SourceContext<String> sourceContext) throws Exception {
                while (isRunning) {
                    //直接通过文件的工具类去读取文件
                    List<String> lines = FileUtils.readLines(
                            new File("input/words.txt"), "utf-8"
                    );
                    //遍历每行的数据，并打印输出，输出每 3s 输出一行
                    for (String line : lines) {
                        sourceContext.collect(line);
                        //休眠3s
                        Thread.sleep(3000);
                    }
                }
            }

            @Override
            public void cancel() {
                isRunning = false;
            }
        });
        // * 4.每行输入数据的单词的拆分
        SingleOutputStreamOperator<Tuple2<String, Integer>> flatMap = node1.flatMap(new FlatMapFunction<String, Tuple2<String, Integer>>() {
            @Override
            public void flatMap(String s, Collector<Tuple2<String, Integer>> collector) throws Exception {
                String[] lines = s.split(",");
                for (String line : lines) {
                    collector.collect(Tuple2.of(line, 1));
                }
            }
        });
        // * 5.根据 word 进行分组/分流
        KeyedStream<Tuple2<String, Integer>, String> keyedStream = flatMap.keyBy(t -> t.f0);
        // * 6.先用 sum 进行求和（自带的算子带状态）
//        SingleOutputStreamOperator<Tuple2<String, Integer>> result = keyedStream.sum(1);
        // * 7.使用
        SingleOutputStreamOperator<Tuple2<String, Integer>> result = keyedStream.flatMap(new RichFlatMapFunction<Tuple2<String, Integer>, Tuple2<String, Integer>>() {
            @Override
            public void flatMap(Tuple2<String, Integer> stringIntegerTuple2, Collector<Tuple2<String, Integer>> collector) throws Exception {
                Tuple2<String, Integer> value = reduceState.value();
                //第一次存的时候，没有值，需要对状态赋值
                if (value == null) {
                    value = stringIntegerTuple2;
                    collector.collect(value);
                    reduceState.update(value);
                }else {
                //更新中间结果
                Tuple2<String, Integer> of = Tuple2.of(value.f0, value.f1 + value.f1);
                reduceState.update(of);
                collector.collect(of);
            }}

            //定义状态的描述器,中间结果状态
            ValueState<Tuple2<String, Integer>> reduceState = null;

            /**
             * 初始化工作
             * 获取ValueState，用于保存或读取中间结果state的值（中间结果值）
             * @param parameters
             * @throws Exception
             */
            @Override
            public void open(Configuration parameters) throws Exception {

                //定义状态的描述器
                //设置生命周期ttl
                StateTtlConfig builder = StateTtlConfig.newBuilder(Time.seconds(6)).setUpdateType(StateTtlConfig.UpdateType.OnReadAndWrite).setStateVisibility(StateTtlConfig.StateVisibility.NeverReturnExpired).build();
                ValueStateDescriptor<Tuple2<String, Integer>> reduceState = new ValueStateDescriptor<>("reduceState", Types.TUPLE(Types.STRING, Types.INT));
                reduceState.enableTimeToLive(builder);
                //从上下文变量获取
                this.reduceState = getRuntimeContext().getState(reduceState);

            }


            @Override
            public void close() throws Exception {
                //System.out.println(reduceState.value().f0 + "--" + reduceState.value().f1);
            }
        });
        // * 8.打印输出的结果
        result.print();
        // * 9.执行流环境
        env.execute();
    }
}

```

broadcaststate

```java
package sz.base.flink.broadcast;

import org.apache.flink.api.common.state.BroadcastState;
import org.apache.flink.api.common.state.MapStateDescriptor;
import org.apache.flink.api.common.state.ReadOnlyBroadcastState;
import org.apache.flink.api.common.typeinfo.Types;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.streaming.api.datastream.BroadcastStream;
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.co.BroadcastProcessFunction;
import org.apache.flink.streaming.api.functions.source.SourceFunction;
import org.apache.flink.util.Collector;

import java.util.*;
import java.util.concurrent.TimeUnit;

/**
 * 公司有10个广告位，其广告的内容（描述和图片）会经常变动（广告到期，更换广告等）
 */
public class BroadcastDemo_4 {
    public static void main(String[] args) throws Exception {
        //1.获取流执行环境
        StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
        //2.设置参数，设置检查点每5s中一次
        env.setParallelism(1);
        env.enableCheckpointing(5000);
        //3构建数据源
        //3.1构建数据源，并将数据转换成整数值类型
        DataStreamSource<String> node1 = env.socketTextStream("node1", 9999);
        //3.2构建自定义数据源用于获取广告位信息（导入数据源）
        DataStreamSource<Map<Integer, Tuple2<String, String>>> mapDataStreamSource = env.addSource(new MySourceForBroadcastFunction());
        //4.将广告位信息广播出去，广播描述信息为 MapStateDescriptor<Integer,Tuple2<String,String>>
        //定义MapStateDescriptor
        MapStateDescriptor<Integer, Tuple2<String, String>> advertiseState = new MapStateDescriptor<>(
                "advertiseState", Types.INT,
                Types.TUPLE(Types.STRING, Types.STRING)
        );
        BroadcastStream<Map<Integer, Tuple2<String, String>>> broadcast = mapDataStreamSource.broadcast(advertiseState);
        //5.将广告ID流connect上广播流
        SingleOutputStreamOperator<Tuple2<String, String>> process = node1.connect(broadcast)
                //6.对关联的数据进行拉宽操作process
                .process(new BroadcastProcessFunction<String, Map<Integer, Tuple2<String, String>>, Tuple2<String, String>>() {
                    //6.1处理每个element
                    @Override
                    public void processElement(String s, ReadOnlyContext readOnlyContext, Collector<Tuple2<String, String>> collector) throws Exception {
                        //通过上下文获取广播状态,把广播出去的获取到
                        ReadOnlyBroadcastState<Integer, Tuple2<String, String>> broadcastState = readOnlyContext.getBroadcastState(advertiseState);
                        //根据value获取配置信息
                        Tuple2<String, String> stringIntegerTuple2 = broadcastState.get(Integer.parseInt(s));
                        //如果配置信息不为空就收集
                        if (stringIntegerTuple2 != null) {
                            collector.collect(stringIntegerTuple2);
                        }
                    }

                    //6.2处理广播element
                    @Override
                    public void processBroadcastElement(Map<Integer, Tuple2<String, String>> integerTuple2Map, Context context, Collector<Tuple2<String, String>> collector) throws Exception {
                        //通过上下文获取广播状态
                        BroadcastState<Integer, Tuple2<String, String>> broadcastState = context.getBroadcastState(advertiseState);
                        //清空
                        broadcastState.clear();
                        //保存最新的
                        broadcastState.putAll(integerTuple2Map);
                    }
                });
        process.print();
        env.execute();


    }

    public static class MySourceForBroadcastFunction implements SourceFunction<Map<Integer, Tuple2<String, String>>> {
        private final Random random = new Random();
        private final List<Tuple2<String, String>> ads = Arrays.asList(
                Tuple2.of("baidu", "搜索引擎"),
                Tuple2.of("google", "科技大牛"),
                Tuple2.of("aws", "全球领先的云平台"),
                Tuple2.of("aliyun", "全球领先的云平台"),
                Tuple2.of("腾讯", "氪金使我变强"),
                Tuple2.of("阿里巴巴", "电商龙头"),
                Tuple2.of("字节跳动", "靠算法出名"),
                Tuple2.of("美团", "黄色小公司"),
                Tuple2.of("饿了么", "蓝色小公司"),
                Tuple2.of("瑞幸咖啡", "就是好喝")
        );
        private boolean isRun = true;

        @Override
        public void run(SourceContext<Map<Integer, Tuple2<String, String>>> ctx) throws Exception {
            while (isRun) {
                Map<Integer, Tuple2<String, String>> map = new HashMap<>();
                int keyCounter = 0;
                for (int i = 0; i < ads.size(); i++) {
                    keyCounter++;
                    map.put(keyCounter, ads.get(random.nextInt(ads.size())));
                }
                ctx.collect(map);

                TimeUnit.SECONDS.sleep(5L);
            }
        }

        @Override
        public void cancel() {
            this.isRun = false;
        }
    }
}

```

端对端仅一次语义

kafka-kafka

```
package sz.base.flink.exatylyonce;

```

```java
import org.apache.flink.api.common.functions.FlatMapFunction;
import org.apache.flink.api.common.restartstrategy.RestartStrategies;
import org.apache.flink.api.common.serialization.SimpleStringSchema;
import org.apache.flink.api.common.time.Time;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.streaming.api.CheckpointingMode;
import org.apache.flink.streaming.api.datastream.DataStreamSource;
import org.apache.flink.streaming.api.datastream.SingleOutputStreamOperator;
import org.apache.flink.streaming.api.environment.CheckpointConfig;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaConsumer;
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaProducer;
import org.apache.flink.streaming.connectors.kafka.KafkaSerializationSchema;
import org.apache.flink.util.Collector;
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.clients.producer.ProducerRecord;

import javax.annotation.Nullable;
import java.util.Properties;
import java.util.Random;

/**
 * 需求：读取kafka中的数据 将数据通过实现一个wordcount 的逻辑 并将其写入到kafka中
 * 要求：支持exactly-once 语义
 * 开发步骤：
 * 1.获取流环境
 * 2.设置checkpoint 1s 状态后端到hdfs或本地file
 * 设置checkpoint属性配置，支持仅一次、超时、并行、容忍、最小间隔、取消任务保存checkpoint
 * 3.设置重启策略3次，10s间隔
 * 4.配置kafka consumer 属性：服务器、消费组、重置从最新、自动发现分区
 * 5.设置consumer设置从最新的读取
 * 6.设置提交offset数据越
 * 7.切分单词并记1 ，遍历每个单词中，随机从0~4中给一个值，如果该值大于3就模拟异常bug，将bug收集
 * 8.对数据进行分组、聚合
 * 9.对最终word和count进行map映射成 word:::count
 * 10设置写到kafka的属性 服务器和事务超时时间5s
 * 11.创建FlinkKafkaProducer
 * 12.将producer 添加到sink ， 需要支持仅一次语义
 * 13.执行流环境
 */
public class FlinkFromKafkaToKafka {
    public static void main(String[] args) throws Exception {
        System.setProperty("HADOOP_USER_NAME","ROOT");
        // * 1.获取流环境
        StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
        // * 2.设置checkpoint 1s 状态后端到hdfs或本地file
        env.setParallelism(1);
        env.enableCheckpointing(1000);
        // * 设置checkpoint属性配置，支持仅一次、超时、并行、容忍、最小间隔、取消任务保存checkpoint
        CheckpointConfig conf = env.getCheckpointConfig();
        conf.setCheckpointStorage("hdfs://node1:8020/flink-checkpoints/FlinkFromKafkaToKafka");
        conf.setCheckpointingMode(CheckpointingMode.EXACTLY_ONCE);
        conf.setCheckpointTimeout(60000);
        conf.setMaxConcurrentCheckpoints(1);
        conf.setTolerableCheckpointFailureNumber(8);
        conf.setMinPauseBetweenCheckpoints(500);
        conf.enableExternalizedCheckpoints(CheckpointConfig.ExternalizedCheckpointCleanup.RETAIN_ON_CANCELLATION);
        // * 3.设置重启策略3次，10s间隔n
        env.setRestartStrategy(RestartStrategies.fixedDelayRestart(3, Time.seconds(10)));
        // * 4.配置kafka consumer 属性：服务器、消费组、重置从最新、自动发现分区
        Properties properties = new Properties();
        properties.setProperty(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "node1:9092,node2:9092,node3:9092");
        properties.setProperty(ConsumerConfig.GROUP_ID_CONFIG, "__consumer_src_topic_");
        properties.setProperty(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, true + "");
        properties.setProperty(ConsumerConfig.AUTO_COMMIT_INTERVAL_MS_CONFIG, 10000 + "");
        properties.setProperty(FlinkKafkaConsumer.KEY_PARTITION_DISCOVERY_INTERVAL_MILLIS, 30 * 60 * 1000 + "");
        //创建读取kafka的实例
        FlinkKafkaConsumer<String> srctopic = new FlinkKafkaConsumer<>("srctopic", new SimpleStringSchema(), properties);
        // * 5.设置consumer设置从最新的读取
        srctopic.setStartFromLatest();
        // 设置提交offset到checkpoint
```

```java
        // * 6.设置提交offset数据源
        srctopic.setCommitOffsetsOnCheckpoints(true);
        // 添加kafka数据源
        DataStreamSource<String> source = env.addSource(srctopic);
        // * 7.切分单词并记1 ， 遍历每个单词中，随机从0~4中给一个值，如果该值大于3就模拟异常bug，将bug收集
        SingleOutputStreamOperator<String> result = source.flatMap(new FlatMapFunction<String, Tuple2<String, Integer>>() {
            Random rm = new Random();

            @Override
            public void flatMap(String s, Collector<Tuple2<String, Integer>> collector) throws Exception {
                String[] words = s.split(" ");
                for (String word : words) {
                    int random = rm.nextInt(5);
                    //模拟一个bug，如果等于4 报错
                    if (random == 4) {
                        throw new Exception("程序除了一点点小bug ， 请检查！ ");
                    }
                    //输出这个tuple2
                    collector.collect(Tuple2.of(word, 1));
                }
            }
        })
                // * 8.对数据进行分组、聚合
                .keyBy(k -> k.f0)
                // * 9.对最终word和count进行map映射成 word:::count
                .map(t -> t.f0 + ":::" + t.f1);
        // * 10设置写到kafka的属性 服务器和事务超时时间5s
        Properties properties1 = new Properties();
        properties1.setProperty(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "node1:9092,node2:9092,node3:9092");
        properties1.setProperty(ProducerConfig.TRANSACTION_TIMEOUT_CONFIG, 2000 + "");
        properties1.setProperty(ProducerConfig.BATCH_SIZE_CONFIG, 5 + "");
        FlinkKafkaProducer<String> producer = new FlinkKafkaProducer<String>(
                "outtopic", new KafkaSerializationSchema<String>() {
            @Override
            public ProducerRecord<byte[], byte[]> serialize(String s, @Nullable Long aLong) {
                return new ProducerRecord<>("outtopic", s.getBytes());
            }
        }, properties1, FlinkKafkaProducer.Semantic.AT_LEAST_ONCE
        );
        // * 11.创建FlinkKafkaProducer
        // * 12.将producer 添加到sink ， 需要支持仅一次语义
        result.addSink(producer);
        // * 13.执行流环境
        env.execute();
    }
}

```

ProcessFunction

实现onTimer方法

```
package sz.base.flink.process;

import org.apache.commons.collections.IteratorUtils;
import org.apache.flink.api.common.functions.MapFunction;
import org.apache.flink.api.common.state.ListState;
import org.apache.flink.api.common.state.ListStateDescriptor;
import org.apache.flink.api.common.time.Time;
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.TimeCharacteristic;
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.streaming.api.functions.KeyedProcessFunction;
import org.apache.flink.util.Collector;

import java.text.SimpleDateFormat;
import java.util.Iterator;

/**
```

```
 * 需求：需要实时监控服务器机架的温度，如果一定时间内温度超过了一定阈值（100度），且后一次上报的温度超过了前一次上报的温度，需要触发告警（温度持续升高中）
 */
public class ServerMonitor {
    public static void main(String[] args) throws Exception {
        //初始化流计算运行环境，制定并行度为1
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(1);
        //开启checkpoint
        env.enableCheckpointing(1000);
        //设置事件时间属性,现在版本已经默认是EventTime
//        env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);
        //接入socket数据源，获取数据 1,100
        DataStreamSource<String> node1 = env.socketTextStream("node1", 9999);
        //将获取到的数据转换成tuple2<Integer,Integer>
        KeyedStream<Tuple2<String, Integer>, String> keyedStream = node1.map(new
MapFunction<String, Tuple2<String, Integer>>() {
            @Override
            public Tuple2<String, Integer> map(String s) throws Exception {
                String[] lines = s.split(",");
                //生成tuple2 ,数据 1,100 2,101 3,103 4,104
                return Tuple2.of(lines[0], Integer.parseInt(lines[1]));
            }
        })
                //根据f0进行分流
                .keyBy(k -> k.f0);
        SingleOutputStreamOperator<String> process = keyedStream.process(new
MyProcessFunction());
        process.printToErr();
        env.execute();
        //实现如下方法
        //2.

    }

    ;

    //自定义processFunction对象，继承KeyedProcessFunction<Tuple,Tuple2<Integer,Integer>,
String>抽象类
```

```
    public static class MyProcessFunction extends KeyedProcessFunction<String, Tuple2<String, Integer>, String> {
        //定义一个变量存储列表中最后一个值
        Integer lastTemperature = 0;
        ListState<Tuple2<String, Integer>> lastTempratureState = null;

        //初始化ListState<Tuple2<机架id，机架温度>>保存上次温度
        //1.open 获取ListState Tuple2<Integer,Integer>获取状态
        //超过100度，并且比上次温度高的数据保存到状态里 ListState
        //状态的数据结构：ValueState ListState BroadcastState ReduceState MapState
        @Override
        public void open(Configuration parameters) throws Exception {
            lastTempratureState = getRuntimeContext().getListState(
                    new ListStateDescriptor<Tuple2<String, Integer>>(
                            "lastTempratureState", Types.TUPLE(Types.STRING, Types.INT)
                    )
            );
        }

        /**
         * 主要实现：读取的每个机架的温度，如果高于100，并且比上次高就给个定时器然后报警，核心业务逻辑
         *
         * @param stringIntegerTuple2
         * @param context
         * @param collector
         * @throws Exception
         */
        @Override
        public void processElement(Tuple2<String, Integer> stringIntegerTuple2, Context context, Collector<String> collector) throws Exception {
            //定义一个时间格式化工具
            SimpleDateFormat sdf =new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
            //当前处理的时间
            long currentTIme = context.timerService().currentProcessingTime();

            Iterable<Tuple2<String, Integer>> temperatures = lastTempratureState.get();
            for (Tuple2<String, Integer> temperature : temperatures) {
                lastTemperature = temperature.f1;
            }
            if (stringIntegerTuple2.f1 > 100 && (stringIntegerTuple2.f1 > lastTemperature)) {
                //将当前的温度先保存到状态,用于下次比较
                lastTempratureState.add(stringIntegerTuple2);
                //获取当前处理时间，注册一个定时器，定时时间为5s
                context.timerService().registerProcessingTimeTimer(currentTIme+5000);
                //返回字符串，打印一下当前的温度和当前的处理时间
                collector.collect( String.format(+stringIntegerTuple2.f1+" 当前的处理时间： "+sdf.format(currentTIme)));
            }else{
                //当前温度 < 100 || 下次温度小于上次的温度
                lastTempratureState.clear();
                //删除触发器
                context.timerService().deleteEventTimeTimer(currentTIme+5000);
                //输出字符串
                collector.collect("当前的告警触发器被解除！");
            }
        }
        //定义触发警告定时器的时长和格式化为：yyyy-MM-dd HH:mm:ss.SSS

        @Override
        public void onTimer(long timestamp, OnTimerContext ctx, Collector<String> out) throws Exception {
            //获取状态中数据size，从状态中获取连续上涨温度有多少次
            Iterator<Tuple2<String, Integer>> iterator = lastTempratureState.get().iterator();
            int size = IteratorUtils.toList(iterator).size();
            System.out.println("当前超过100度并累加温度升高的个数为："+size);
            if(size>1){
                System.out.println("当前温度过高，高温报警，滴滴滴！");
                out.collect("当前温度过高，高温报警，滴滴滴！");
            }
            //清空历史数据
            lastTempratureState.clear();
        }
    }

    ;
}

```

双流JOIN

窗口实现

```java
package sz.base.flink.join;

import com.alibaba.fastjson.JSON;
import lombok.Data;
import org.apache.flink.api.common.eventtime.*;
import org.apache.flink.api.common.functions.JoinFunction;
import org.apache.flink.configuration.Configuration;
import org.apache.flink.streaming.api.datastream.SingleOutputStreamOperator;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.functions.source.RichSourceFunction;
import org.apache.flink.streaming.api.windowing.assigners.SlidingProcessingTimeWindows;
import org.apache.flink.streaming.api.windowing.assigners.TumblingProcessingTimeWindows;
import org.apache.flink.streaming.api.windowing.time.Time;

import java.math.BigDecimal;
import java.util.ArrayList;
import java.util.List;
import java.util.Random;
import java.util.UUID;
import java.util.concurrent.TimeUnit;

/**
 * 需求-使用两个指定source模拟数据，一个source是订单明细，一个source是商品数据，我们通过
window join，将数据关联到一起
 * window join
 *
 *
 */
public class DoubleStreamJoin {
    public static void main(String[] args) throws Exception {
        //获取流执行环境
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(1);
        //添加数据源，商品表和订单表
        SingleOutputStreamOperator<Goods> goodsDataStream = env.addSource(new GoodsSource()).assignTimestampsAndWatermarks(new GoodsWatermark());
        //订单项数据流
        SingleOutputStreamOperator<OrderItem> orderDataStream = env.addSource(new OrderItemSource()).assignTimestampsAndWatermarks(new OrderItemWatermark());
        //分配水印机制
        //进行双流join（滚动事件时间窗口）
orderDataStream.join(goodsDataStream).where(OrderItem::getGoodsId).equalTo(Goods::getGoodsId)
                //滚动窗口
//                  .window(TumblingProcessingTimeWindows.of(Time.seconds(5)))
                //滑动窗口
.window(SlidingProcessingTimeWindows.of(Time.seconds(5),Time.seconds(1)))
                .apply(new JoinFunction<OrderItem, Goods, FactOrderItem>() {
            @Override
            public FactOrderItem join(OrderItem orderItem, Goods goods) throws Exception {
                FactOrderItem factOrderItem = new FactOrderItem();
                factOrderItem.setGoodsId(goods.goodsId);
                factOrderItem.setGoodsName(goods.goodsName);
                factOrderItem.setCount(new BigDecimal(orderItem.count));
                factOrderItem.setTotalMoney(goods.getGoodsPrice().multiply(new BigDecimal(orderItem.count)));
                return factOrderItem;
            }
        })
        //打印结果
        .printToErr();
        //执行流环境
        env.execute();


    }
    //商品类
    @Data
    public static class Goods {
        private String goodsId;
        private String goodsName;
        private BigDecimal goodsPrice;

```

```java
        public static List<Goods> GOODS_LIST;
        public static Random r;

        static  {
            r = new Random();
            GOODS_LIST = new ArrayList<>();
            GOODS_LIST.add(new Goods("1", "小米12", new BigDecimal(4890)));
            GOODS_LIST.add(new Goods("2", "iphone12", new BigDecimal(12000)));
            GOODS_LIST.add(new Goods("3", "MacBookPro", new BigDecimal(15000)));
            GOODS_LIST.add(new Goods("4", "Thinkpad X1", new BigDecimal(9800)));
            GOODS_LIST.add(new Goods("5", "MeiZu One", new BigDecimal(3200)));
            GOODS_LIST.add(new Goods("6", "Mate 40", new BigDecimal(6500)));
        }

        public static Goods randomGoods() {
            int rIndex = r.nextInt(GOODS_LIST.size());
            return GOODS_LIST.get(rIndex);
        }

        public Goods() {
        }

        public Goods(String goodsId, String goodsName, BigDecimal goodsPrice) {
            this.goodsId = goodsId;
            this.goodsName = goodsName;
            this.goodsPrice = goodsPrice;
        }

        @Override
        public String toString() {
            return JSON.toJSONString(this);
        }
    }

    //订单明细类
    @Data
    public static class OrderItem {
        private String itemId;
        private String goodsId;
        private Integer count;

        @Override
        public String toString() {
            return JSON.toJSONString(this);
        }
    }

    //关联结果
    @Data
    public static class FactOrderItem {
        private String goodsId;
        private String goodsName;
        private BigDecimal count;
        private BigDecimal totalMoney;
        @Override
        public String toString() {
            return JSON.toJSONString(this);
        }
    }
    //构建一个商品Stream源（这个好比就是维表）
    public static class GoodsSource extends RichSourceFunction<Goods> {
        private Boolean isCancel;
        @Override
        public void open(Configuration parameters) throws Exception {
            isCancel = false;
        }
        @Override
        public void run(SourceContext sourceContext) throws Exception {
            while(!isCancel) {
                Goods.GOODS_LIST.stream().forEach(goods ->
sourceContext.collect(goods));
                TimeUnit.SECONDS.sleep(1);
            }
        }
        @Override
        public void cancel() {
            isCancel = true;
        }
    }
    //构建订单明细Stream源
    public static class OrderItemSource extends RichSourceFunction<OrderItem> {
        private Boolean isCancel;
        private Random r;
        @Override
        public void open(Configuration parameters) throws Exception {
            isCancel = false;
            r = new Random();
        }
        @Override
        public void run(SourceContext sourceContext) throws Exception {
            while(!isCancel) {
                Goods goods = Goods.randomGoods();
                OrderItem orderItem = new OrderItem();
                orderItem.setGoodsId(goods.getGoodsId());
                orderItem.setCount(r.nextInt(10) + 1);
                orderItem.setItemId(UUID.randomUUID().toString());
                sourceContext.collect(orderItem);
                orderItem.setGoodsId("111");
                sourceContext.collect(orderItem);
                TimeUnit.SECONDS.sleep(1);
            }
        }

        @Override
        public void cancel() {
            isCancel = true;
        }
    }
    //构建水印分配器（此处为了简单），直接使用系统时间了
    public static class GoodsWatermark implements WatermarkStrategy<Goods> {

        @Override
        public TimestampAssigner<Goods>
createTimestampAssigner(TimestampAssignerSupplier.Context context) {
            return (element, recordTimestamp) -> System.currentTimeMillis();
        }

        @Override
        public WatermarkGenerator<Goods>
createWatermarkGenerator(WatermarkGeneratorSupplier.Context context) {
```

```java
            return new WatermarkGenerator<Goods>() {
                @Override
                public void onEvent(Goods event, long eventTimestamp, WatermarkOutput output) {
                    output.emitWatermark(new Watermark(System.currentTimeMillis()));
                }

                @Override
                public void onPeriodicEmit(WatermarkOutput output) {
                    output.emitWatermark(new Watermark(System.currentTimeMillis()));
                }
            };
        }
    }

    public static class OrderItemWatermark implements WatermarkStrategy<OrderItem> {
        @Override
        public TimestampAssigner<OrderItem> createTimestampAssigner(TimestampAssignerSupplier.Context context) {
            return (element, recordTimestamp) -> System.currentTimeMillis();
        }
        @Override
        public WatermarkGenerator<OrderItem> createWatermarkGenerator(WatermarkGeneratorSupplier.Context context) {
            return new WatermarkGenerator<OrderItem>() {
                @Override
                public void onEvent(OrderItem event, long eventTimestamp, WatermarkOutput output) {
                    output.emitWatermark(new Watermark(System.currentTimeMillis()));
                }
                @Override
                public void onPeriodicEmit(WatermarkOutput output) {
                    output.emitWatermark(new Watermark(System.currentTimeMillis()));
                }
            };
        }
    }
}

```

Interval join

```java
package sz.base.flink.join;

import com.alibaba.fastjson.JSON;
import lombok.Data;
import org.apache.flink.api.common.eventtime.*;
import org.apache.flink.configuration.Configuration;
import org.apache.flink.streaming.api.datastream.SingleOutputStreamOperator;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.functions.co.ProcessJoinFunction;
import org.apache.flink.streaming.api.functions.source.RichSourceFunction;
import org.apache.flink.streaming.api.windowing.time.Time;
import org.apache.flink.util.Collector;

import java.math.BigDecimal;
import java.util.ArrayList;
import java.util.List;
import java.util.Random;
import java.util.UUID;
import java.util.concurrent.TimeUnit;

/**
 * 双流join-其中一个流指定一个上界和下界，和另外一个数据进行关联操作
 */
public class IntervalStreamJoin {
    public static void main(String[] args) throws Exception {
        //获取流执行环境
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
        //设置并行度
        env.setParallelism(1);
        //构建两个数据流
        SingleOutputStreamOperator<Goods> goodsDateStream = env.addSource(new
GoodsSource11()).assignTimestampsAndWatermarks(new GoodsWatermark());
        SingleOutputStreamOperator<OrderItem> orderDataStream = env.addSource(new
OrderItemSource()).assignTimestampsAndWatermarks(new OrderItemWatermark());
        //双流interval join，计算前两秒和下一秒
        SingleOutputStreamOperator<FactOrderItem> process = orderDataStream.keyBy(t ->
t.getGoodsId()).intervalJoin(goodsDateStream.keyBy(k ->
k.getGoodsId())).between(Time.seconds(-2), Time.seconds(1))
//                不包含上界[-2,)
                .upperBoundExclusive()
                //不包含下界两个一起用则都不包含
//                .lowerBoundExclusive()
                //进行process处理
                .process(new ProcessJoinFunction<OrderItem, Goods, FactOrderItem>() {
                    @Override
                    public void processElement(OrderItem orderItem, Goods goods,
Context context, Collector<FactOrderItem> collector) throws Exception {
                        FactOrderItem factOrderItem = new FactOrderItem();
                        factOrderItem.setGoodsId(goods.getGoodsId());
                        factOrderItem.setCount(new BigDecimal(orderItem.count));
                        factOrderItem.setGoodsName(goods.goodsName);
                        factOrderItem.setTotalMoney(goods.getGoodsPrice().multiply(new
BigDecimal(orderItem.getCount())));
                        collector.collect(factOrderItem);
                    }
                });
        process.printToErr();
        env.execute();
    }

    //商品类
    @Data
    public static class Goods {
        private String goodsId;
        private String goodsName;
        private BigDecimal goodsPrice;

        public static List<Goods> GOODS_LIST;
        public static Random r;

        static {
            r = new Random();
            GOODS_LIST = new ArrayList<>();
            GOODS_LIST.add(new Goods("1", "小米12", new BigDecimal(4890)));
            GOODS_LIST.add(new Goods("2", "iphone12", new BigDecimal(12000)));
            GOODS_LIST.add(new Goods("3", "MacBookPro", new BigDecimal(15000)));
            GOODS_LIST.add(new Goods("4", "Thinkpad X1", new BigDecimal(9800)));
            GOODS_LIST.add(new Goods("5", "MeiZu One", new BigDecimal(3200)));
            GOODS_LIST.add(new Goods("6", "Mate 40", new BigDecimal(6500)));
        }

        public static Goods randomGoods() {
            int rIndex = r.nextInt(GOODS_LIST.size());
            return GOODS_LIST.get(rIndex);
        }

        public Goods() {
        }

        public Goods(String goodsId, String goodsName, BigDecimal goodsPrice) {
            this.goodsId = goodsId;
            this.goodsName = goodsName;
            this.goodsPrice = goodsPrice;
        }

        @Override
        public String toString() {
            return JSON.toJSONString(this);
        }
    }

    //订单明细类
    @Data
    public static class OrderItem {
        private String itemId;
        private String goodsId;
        private Integer count;

        @Override
        public String toString() {
            return JSON.toJSONString(this);
        }
    }

    //关联结果
    @Data
    public static class FactOrderItem {
        private String goodsId;
        private String goodsName;
        private BigDecimal count;
        private BigDecimal totalMoney;

        @Override
        public String toString() {
            return JSON.toJSONString(this);
        }
    }

    //构建一个商品Stream源（这个好比就是维表）
    public static class GoodsSource11 extends RichSourceFunction<Goods> {
        private Boolean isCancel;

        @Override
        public void open(Configuration parameters) throws Exception {
            isCancel = false;
        }

        @Override
        public void run(SourceContext sourceContext) throws Exception {
            while (!isCancel) {
                Goods.GOODS_LIST.stream().forEach(goods ->
sourceContext.collect(goods));
                TimeUnit.SECONDS.sleep(1);
            }
        }

        @Override
        public void cancel() {
            isCancel = true;
        }
    }

    //构建订单明细Stream源
    public static class OrderItemSource extends RichSourceFunction<OrderItem> {
        private Boolean isCancel;
        private Random r;
```

```java

        @Override
        public void open(Configuration parameters) throws Exception {
            isCancel = false;
            r = new Random();
        }

        @Override
        public void run(SourceContext sourceContext) throws Exception {
            while (!isCancel) {
                Goods goods = Goods.randomGoods();
                OrderItem orderItem = new OrderItem();
                orderItem.setGoodsId(goods.getGoodsId());
                orderItem.setCount(r.nextInt(10) + 1);
                orderItem.setItemId(UUID.randomUUID().toString());
                sourceContext.collect(orderItem);
                orderItem.setGoodsId("111");
                sourceContext.collect(orderItem);
                TimeUnit.SECONDS.sleep(1);
            }
        }

        @Override
        public void cancel() {
            isCancel = true;
        }
    }

    //构建水印分配器（此处为了简单），直接使用系统时间了
    public static class GoodsWatermark implements WatermarkStrategy<Goods> {

        @Override
        public TimestampAssigner<Goods>
createTimestampAssigner(TimestampAssignerSupplier.Context context) {
            return (element, recordTimestamp) -> System.currentTimeMillis();
        }

        @Override
        public WatermarkGenerator<Goods>
createWatermarkGenerator(WatermarkGeneratorSupplier.Context context) {
```

```java
            return new WatermarkGenerator<Goods>() {
                @Override
                public void onEvent(Goods event, long eventTimestamp, WatermarkOutput output) {
                    output.emitWatermark(new Watermark(System.currentTimeMillis()));
                }

                @Override
                public void onPeriodicEmit(WatermarkOutput output) {
                    output.emitWatermark(new Watermark(System.currentTimeMillis()));
                }
            };
        }
    }

    public static class OrderItemWatermark implements WatermarkStrategy<OrderItem> {
        @Override
        public TimestampAssigner<OrderItem> createTimestampAssigner(TimestampAssignerSupplier.Context context) {
            return (element, recordTimestamp) -> System.currentTimeMillis();
        }

        @Override
        public WatermarkGenerator<OrderItem> createWatermarkGenerator(WatermarkGeneratorSupplier.Context context) {
            return new WatermarkGenerator<OrderItem>() {
                @Override
                public void onEvent(OrderItem event, long eventTimestamp, WatermarkOutput output) {
                    output.emitWatermark(new Watermark(System.currentTimeMillis()));
                }

                @Override
                public void onPeriodicEmit(WatermarkOutput output) {
                    output.emitWatermark(new Watermark(System.currentTimeMillis()));
                }
            };
        }
    }
}

```

FlinkTableAPI和FlinkSQL

FlinkSQL

```java
package sz.base.flink.cases;


import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.datastream.DataStreamSource;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.table.api.DataTypes;
import org.apache.flink.table.api.EnvironmentSettings;
import org.apache.flink.table.api.Schema;
import org.apache.flink.table.api.Table;
import org.apache.flink.table.api.bridge.java.StreamTableEnvironment;
import org.apache.flink.types.Row;


import java.beans.Expression;

import static org.apache.flink.table.api.Expressions.$;


/**
 * 案例：单词统计的案例，使用FlinkTable & FlinkSQL
 * 输出表的样式：
 * Hello | 2
 * word | 1
 *
 * sql 的实现样式
 * select word,sum(1) as frequency from WC group by word ;
 *
 * 总结：
 * 数据流：DataStream
 * 虚拟表："t_words"
 * Table对象 result
 * Table对象 -> DataStream -> 打印输出
 */
public class WordCountDemo {
    public static void main(String[] args) throws Exception {
        //1.获取流环境
        StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
        //环境设置
        EnvironmentSettings settings = EnvironmentSettings.newInstance()
                .useBlinkPlanner().inStreamingMode().build();
        //设置流表环境
        StreamTableEnvironment tEnv = StreamTableEnvironment.create(env, settings);
        //2.source获取 单词信息
        DataStreamSource<WC> input = env.fromElements(new WC("hello", 1),
                new WC("world", 1),
                new WC("flink", 3),
                new WC("hadoop", 2),
                new WC("hello", 2));
        //3.创建视图wordcount,参数：表名 数据流 字段...
//        tEnv.createTemporaryView("t_words",input,$("word"),$("frequency"));
        //参数：表名，数据流，schema(构造者模式)
        tEnv.createTemporaryView("t_words",input, Schema.newBuilder().column("word", DataTypes.STRING()).column("frequency",DataTypes.BIGINT()).build() );
        //4.执行查询，单词统计
        Table result = tEnv.sqlQuery("select word,sum(frequency) as cnt from t_words " +
                "group by word");
        //5.输出结果retractStream获取数据流
        //打印当前表的表结构
        result.printSchema();
        //将Table对象转换成 DataStream 在输出
        DataStream<Tuple2<Boolean, Row>> dataStream = tEnv.toRetractStream(result, Row.class);
        //打印输出结果
        dataStream.printToErr();
        //执行流环境
        env.execute();
    }

    @Data
    @AllArgsConstructor
    @NoArgsConstructor
    public static class WC{
        public String word;
        public long frequency;
    }
}

```

FlinkTableAPI

```
package sz.base.flink.cases;


import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.datastream.DataStreamSource;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.table.api.DataTypes;
import org.apache.flink.table.api.EnvironmentSettings;
import org.apache.flink.table.api.Schema;
import org.apache.flink.table.api.Table;
import org.apache.flink.table.api.bridge.java.StreamTableEnvironment;
import org.apache.flink.types.Row;

import static org.apache.flink.table.api.Expressions.$;


/**
 * 案例：单词统计的案例，使用FlinkTable
 * 输出表的样式：
 * Hello | 2
 * word | 1
 *
 * sql 的实现样式
 * select word,sum(1) as frequency from WC group by word ;
 *
 * 总结:
 * 数据流: DataStream -> 虚拟表 path
 * 虚拟表: "t_words" ->  Table对象
 * Table API -> result
 * Table对象 -> DataStream -> 打印输出
 */
public class WordCountTableDemo {
    public static void main(String[] args) throws Exception {
        //1.获取流环境
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
        //环境设置
        EnvironmentSettings settings = EnvironmentSettings.newInstance()
                .useBlinkPlanner().inStreamingMode().build();
        //设置流表环境
        StreamTableEnvironment tEnv = StreamTableEnvironment.create(env, settings);
        //2.source获取 单词信息
        DataStreamSource<WC> input = env.fromElements(new WC("hello", 1),
                new WC("world", 1),
                new WC("flink", 3),
                new WC("hadoop", 2),
                new WC("hello", 2));
        //3.创建视图wordcount,参数: 表名 数据流 字段...
//          tEnv.createTemporaryView("t_words",input,$("word"),$("frequency"));
        //参数: 表名, 数据流, schema(构造者模式)
        tEnv.createTemporaryView("t_words",input, Schema.newBuilder().column("word",
DataTypes.STRING()).column("frequency",DataTypes.BIGINT()).build() );
        //4.执行查询, 单词统计
        //虚拟表转换成Table对象
        Table words = tEnv.from("t_words");
        //使用Table API 实现wordcount
        Table result = words.groupBy($("word")).select($("word"),
$("frequency").sum().as("cnt"));
        //5.输出结果retractStream获取数据流
        //打印当前表的表结构

        result.printSchema();
        //将Table对象转换成 DataStream 在输出
```

```java
        DataStream<Tuple2<Boolean, Row>> dataStream = tEnv.toRetractStream(result, Row.class);
        //打印输出结果
        dataStream.printToErr();
        //执行流环境
        env.execute();
    }

    @Data
    @AllArgsConstructor
    @NoArgsConstructor
    public static class WC{
        public String word;
        public long frequency;
    }
}

```

输出到表

- **输出到文件系统**

```java
package sz.base.flinkconnector;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.apache.flink.api.common.functions.MapFunction;
import org.apache.flink.api.common.restartstrategy.RestartStrategies;
import org.apache.flink.api.java.io.TextInputFormat;
import org.apache.flink.streaming.api.datastream.DataStream;
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.source.FileProcessingMode;
import org.apache.flink.table.api.EnvironmentSettings;
import org.apache.flink.table.api.Table;
import org.apache.flink.table.api.bridge.java.StreamTableEnvironment;

import static org.apache.flink.table.api.Expressions.$;
```

```java

/**
 * 需求- 将过滤出来的地区为北京地区的数据写出到文件系统 FileSystem 中
 */
public class OrderSinkFileSystem {
    public static void main(String[] args) throws Exception {

        //创建流执行环境设置并行度
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(1);
        //设置环境设置为blink计划器并且是流模式
        EnvironmentSettings settings =
EnvironmentSettings.newInstance().useBlinkPlanner().inStreamingMode().build();
        //谁次checkpoint
        env.enableCheckpointing(10000);
        //设置重启策略
        env.setRestartStrategy(RestartStrategies.fixedDelayRestart(3, 3000));
        //从 order.csv 读取文件数据源 读取为流数据
        DataStreamSource<String> source = env.readFile(new TextInputFormat(null),
"input/order.csv", FileProcessingMode.PROCESS_CONTINUOUSLY, 60 * 1000);
        //将字符串map转换切分转换成 OrderInfo 对象
        SingleOutputStreamOperator<OrderInfo> map = source.map(new MapFunction<String,
OrderInfo>() {
            @Override
            public OrderInfo map(String s) throws Exception {
                String[] arr = s.split(",");
                OrderInfo orderInfo = new OrderInfo(arr[0]
                        , Long.parseLong(arr[1]),
                        arr[2],
                        Double.parseDouble(arr[3])
                        , arr[4]);
                return orderInfo;
            }
        });
        //创建表环境
        StreamTableEnvironment tEnv = StreamTableEnvironment.create(env, settings);
        //将数据流转换成 Table 数据流转table，from只是把路径转换成table
        Table orderTable = tEnv.fromDataStream(map, $("uid"), $("tms"), $("category"),
$("price"), $("areaName"));
        //1. Flink Table ap    i 语法筛选filter出区域为北京的所有字段数据
```

```java
/*        Table result = orderTable.where(
                $("areaName").isEqual("北京")
        ).select($("uid"), $("tms"), $("category"), $("price"), $("areaName"));*/
        //2. 将table创建临时视图
        tEnv.createTemporaryView("t_order", orderTable);

        //编写SQL查询获取北京的所有字段信息
        Table result2Table = tEnv.sqlQuery("select * from t_order where areaName='北京'
");
        //1.1将结果Table转换成数据流
//        DataStream<Tuple2<Boolean, OrderInfo>> result1 = tEnv.toRetractStream(result,
OrderInfo.class);
        //2.1
        DataStream<OrderInfo> result1 = tEnv.toAppendStream(result2Table,
OrderInfo.class);
        //打印输出到文件系统 FileSystem 中
        String sql = "CREATE TABLE t_order_result (" +
                " uid STRING," +
                " tms bigint," +
                " category STRING," +
                " price double," +
                " areaName STRING" +
                ") WITH( " +
                " 'connector'='filesystem'," +
                " 'path'='file:///D:/order'," +
                " 'format'='csv'," +
                " 'sink.rolling-policy.rollover-interval'='1 min' " +
                ")";
        //执行流环境
        tEnv.executeSql(sql);
        result2Table.executeInsert("t_order_result");
        env.execute();

    }

    @Data
    @AllArgsConstructor
    @NoArgsConstructor
    public static class OrderInfo {
        private String uid;
        private Long tms;
        private String category;
        private Double price;
        private String areaName;
    }
}

```

输出到kafka

```java
package sz.base.flinkconnector;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.apache.flink.api.common.functions.MapFunction;
import org.apache.flink.api.common.restartstrategy.RestartStrategies;
import org.apache.flink.api.java.io.TextInputFormat;
import org.apache.flink.streaming.api.datastream.DataStream;
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.source.FileProcessingMode;
import org.apache.flink.table.api.EnvironmentSettings;
import org.apache.flink.table.api.Table;
import org.apache.flink.table.api.bridge.java.StreamTableEnvironment;

import static org.apache.flink.table.api.Expressions.$;

/**
 * 需求 - 筛选出地区为北京的数据并将其写入kafka
 */
public class OrderSinkFKafka {
    public static void main(String[] args) throws Exception {

        //创建流执行环境设置并行度
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(1);
        //设置环境设置为blink计划器并且是流模式
        EnvironmentSettings settings = EnvironmentSettings.newInstance().useBlinkPlanner().inStreamingMode().build();
        //谁次checkpoint
        env.enableCheckpointing(10000);
        //设置重启策略
        env.setRestartStrategy(RestartStrategies.fixedDelayRestart(3, 3000));
        //从 order.csv 读取文件数据源 读取为流数据
        DataStreamSource<String> source = env.readFile(new TextInputFormat(null), "input/order.csv", FileProcessingMode.PROCESS_CONTINUOUSLY, 60 * 1000);
        //将字符串map转换切分转换成 OrderInfo 对象
        SingleOutputStreamOperator<OrderSinkFileSystem.OrderInfo> map = source.map(new MapFunction<String, OrderSinkFileSystem.OrderInfo>() {
          @Override
          public OrderSinkFileSystem.OrderInfo map(String s) throws Exception {
            String[] arr = s.split(",");
            OrderSinkFileSystem.OrderInfo orderInfo = new OrderSinkFileSystem.OrderInfo(arr[0]
                , Long.parseLong(arr[1]),
                arr[2],
                Double.parseDouble(arr[3])
                , arr[4]);
            return orderInfo;
          }
        });
        //创建表环境
        StreamTableEnvironment tEnv = StreamTableEnvironment.create(env, settings);
        //将数据流转换成 Table 数据流转table, from只是把路径转换成table
        Table orderTable = tEnv.fromDataStream(map, $("uid"), $("tms"), $("category"), $("price"), $("areaName"));
        //1. Flink Table ap   i 语法筛选filter出区域为北京的所有字段数据
/*   Table result = orderTable.where(
            $("areaName").isEqual("北京")
        ).select($("uid"), $("tms"), $("category"), $("price"), $("areaName"));*/
        //2. 将table创建临时视图
        tEnv.createTemporaryView("t_order", orderTable);

        //编写SQL查询获取北京的所有字段信息
        Table result2Table = tEnv.sqlQuery("select * from t_order where areaName='北京'");
        //1.1将结果Table转换成数据流
//   DataStream<Tuple2<Boolean, OrderInfo>> result1 = tEnv.toRetractStream(result, OrderInfo.class);
```

```java
        //2.1
        DataStream<OrderSinkFileSystem.OrderInfo> result1 =
tEnv.toAppendStream(result2Table, OrderSinkFileSystem.OrderInfo.class);
        //打印输出到文件系统 FileSystem 中
        String sql = "CREATE TABLE t_order_result (" +
                " uid STRING," +
                " tms bigint," +
                " category STRING," +
                " price double," +
                " areaName STRING" +
                ") WITH( " +
                " 'connector'='kafka'," +
                " 'topic'='output'," +
                " 'properties.bootstrap.servers'='node1:9092,node2:9092,node3:9092' ," +
                " 'format'='json' ," +
                " 'scan.topic.partition-discovery.interval'='30000' ," +
                " 'sink.semantic'='at-least-once' , " +
                " 'scan.startup.mode'='latest-offset' , " +
                " 'properties.group.id'='__consumer_output_' " +
                ")";
        //执行流环境
        tEnv.executeSql(sql);
        result2Table.executeInsert("t_order_result");
        env.execute();

    }

    @Data
    @AllArgsConstructor
    @NoArgsConstructor
    public static class OrderInfo {
        private String uid;
        private Long tms;
        private String category;
        private Double price;
        private String areaName;
    }
}
```

kafka输出到MySQL

```java
package sz.base.flinkconnector;


import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.table.api.EnvironmentSettings;
import org.apache.flink.table.api.Table;
import org.apache.flink.table.api.bridge.java.StreamTableEnvironment;
import org.apache.flink.types.Row;

/**
 * 需求 - 将kafka中的订单的数据写入到MySQL中
 * 开发步骤:
 * 1.在MySQL中先创建数据库和数据表
 * 2.创建流表环境
 * 3.读取kafka中的数据源
 * 4执行kafka数据源SQL
 * 5.写入MySQL的SQL并执行
 * 6.实现 insert into 目标表 select 字段列表 from 源表
 * 7.执行流表环境
 */
public class OrderKafkaMySQL {

    public static void main(String[] args) throws Exception {
        // * 1.在MySQL中先创建数据库和数据表
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
        // * 2.创建流表环境
        EnvironmentSettings settings =
EnvironmentSettings.newInstance().useBlinkPlanner().inStreamingMode().build();
        StreamTableEnvironment tEnv = StreamTableEnvironment.create(env, settings);
        // * 3.读取kafka中的数据源
        String kafkaTable = "CREATE TABLE KafkaTable (" +
                "  `uid` STRING," +
                "  `tms` BIGINT," +
                "  `category` STRING," +
                "  `price` DOUBLE," +
                "  `areaName` STRING" +
                ") WITH (" +
                " 'connector'='kafka'," +
                " 'topic'='output'," +
                " 'properties.bootstrap.servers'='node1:9092,node2:9092,node3:9092' ," +
                " 'format'='json' ," +
                " 'scan.topic.partition-discovery.interval'='30000' ," +
                " 'sink.semantic'='at-least-once' , " +
                " 'scan.startup.mode'='earliest-offset' , " +
                " 'properties.group.id'='__consumer_output_' " +
                ")";
        // * 4执行kafka数据源SQL,虚拟表就会被创建
        tEnv.executeSql(kafkaTable);
        //读取kafka中的数据并打印输出
        Table table1 = tEnv.from("KafkaTable");
        tEnv.toAppendStream(table1, Row.class).printToErr();
        // * 5.写入MySQL的SQL并执行
        String mysqlTable = "CREATE TABLE MyUserTable (" +
                "  uid STRING," +
                "  tms BIGINT," +
                "  category STRING," +
                "  price DOUBLE," +
                "  areaName STRING," +
                "  PRIMARY KEY (uid) NOT ENFORCED" +
                ") WITH (" +
                "   'connector' = 'jdbc'," +
                "   'url' = 'jdbc:mysql://localhost:3306/test?useSSL=false&characterEncoding=utf-8'," +
                "   'username'='root'," +
                "   'password'='root'," +
                "   'sink.buffer-flush.max-rows'='1'," +
                "   'sink.buffer-flush.interval'='1s'," +
                "   'table-name' = 'order_test'" +
                ");";
        //执行落地表
        tEnv.executeSql(mysqlTable);
        // * 6.实现 insert into 目标表 select 字段列表 from 源表
        tEnv.executeSql("INSERT INTO order_test select uid,tms,category,price,areaName from KafkaTable");
        // * 7.执行流表环境
        env.execute();
```

```java
    }
}

```

- **分配水印**

```java
package sz.base.flik.sql;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.apache.calcite.avatica.com.google.protobuf.SourceContext;
import org.apache.flink.api.common.eventtime.WatermarkStrategy;
import org.apache.flink.api.common.restartstrategy.RestartStrategies;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.datastream.SingleOutputStreamOperator;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.functions.source.RichSourceFunction;
import org.apache.flink.table.api.EnvironmentSettings;
import org.apache.flink.table.api.Table;
import org.apache.flink.table.api.bridge.java.StreamTableEnvironment;
import org.apache.flink.types.Row;

import java.time.Duration;
import java.util.Random;
import java.util.UUID;
import java.util.concurrent.TimeUnit;

import static org.apache.flink.table.api.Expressions.$;
import static org.apache.flink.table.api.Expressions.e;

/**
 * 需求   - 实现一个订单需求案例
 * 随机生成一个订单并将其转换，根据事件时间进行分组，求出订单的和
 */
public class OrderTime {
    public static void main(String[] args) throws Exception {
        //1.创建流执行环境和流表环境
        StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
        EnvironmentSettings settings = EnvironmentSettings.newInstance().useBlinkPlanner().inStreamingMode().build();
        //开启Checkpoint
        env.enableCheckpointing(1000);
        env.setParallelism(1);
        //开启重启策略
        env.setRestartStrategy(RestartStrategies.fixedDelayRestart(3,3000));
        StreamTableEnvironment tEnv = StreamTableEnvironment.create(env, settings);
        //2.source 自定义order 每1s 中睡眠一次
        //3.Transformation 分配时间戳水印2s
        SingleOutputStreamOperator<Order> source = env.addSource(new MySource()).assignTimestampsAndWatermarks(WatermarkStrategy.<Order>forBoundedOutOfOrderness(Duration.ofSeconds(3)).withTimestampAssigner((k, t) -> k.createTime));
        //4.注册表，创建临时视图并分配rowtime
        tEnv.createTemporaryView("t_order",source,
                $("orderId"),
                $("userId"),
                $("money"),
                $("createTime").rowtime()//标记当前的字段是事件时间
        );
        //获取流表对象
        //注册临时表
        //5.编写SQL，根据userid 和createTime 滚动分组统计userid、订单总笔数、最大、最小金额
        Table result = tEnv.sqlQuery("select userId,count(orderId)as cnt , max(money) as maxMoney, min(money) as minMoney" +
                " from t_order" +
                " group by userId,tumble(createTime,interval '5' second)"//根据userid和时间5s分组
        );
        //6.执行查询语句返回结果,查询执行计划
        System.out.println(result.explain());
        //7.sink toRetractStream → 将计算后的新的数据在DataStream原数据的基础上更新true或是删除false
        DataStream<Tuple2<Boolean, Row>> tuple2DataStream = tEnv.toRetractStream(result, Row.class);
        tuple2DataStream.printToErr();
        env.execute();

    }
    private static class MySource extends RichSourceFunction<Order> {
        //循环条件
        volatile boolean isRunning = true;

        @Override
        public void run(SourceContext<Order> ctx) throws Exception {
            Random rm = new Random();
            while (isRunning) {
                Order order = new Order();
                order.setOrderId(UUID.randomUUID().toString());
                order.setUserId(rm.nextInt(3));
                order.setMoney(rm.nextInt(101));
                //水印机制，模拟延迟的数据，随机三秒
                order.setCreateTime(System.currentTimeMillis() - rm.nextInt(3) * 1000);
                ctx.collect(order);
                //一秒一条
                TimeUnit.SECONDS.sleep(1);
            }
        }

        @Override
        public void cancel() {
            isRunning = false;
        }
    }

    @Data
    @AllArgsConstructor
    @NoArgsConstructor
    public static class Order {
        private String orderId;
        private Integer userId;
        private Integer money;
        private Long createTime;
    }
}

```