### window function

在确定窗口类型之后，便可以定义窗口数据的计算逻辑，也就是定义Window Function。Flink提供了四种类型Window Function，其中有ReduceFunction、AggregateFunction、FlodFunction和ProcessWindowFunction。其中ReduceFunction、AggregateFunction、FlodFunction根据计算原理，属于增量聚合函数，而ProcessWindowFunction属于全量聚合函数。增量聚合函数是基于中间状态计算结果的，窗口中只维护中间状态结果值，不需要缓存原始的数据，而全量窗口函数在窗口触发时对所有的原始数据进行汇总计算，因此相对性能会较差。

### ReduceFunction

ReduceFunction：对输入的两个相同类型的数据元素按照指定的计算方法进行聚合，然后输出一个类型相同的结果元素

开发：

**import** org.apache.flink.api.common.functions.ReduceFunction;  
**import** org.apache.flink.api.java.tuple.Tuple2;  
**import** org.apache.flink.streaming.api.datastream.DataStream;  
**import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;  
**import** org.apache.flink.streaming.api.windowing.time.Time;  
  
**import** java.util.ArrayList;  
**import** java.util.List;  
  
**public class** wr1 {  
 **public static void** main(String[] args) **throws** Exception {  
 StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  
 List source1 = **new** ArrayList();  
 source1.add(**new** Tuple2<String,Long>(**"sq"**,15L));  
 source1.add(**new** Tuple2<String,Long>(**"sq"**,10L));  
 source1.add(**new** Tuple2<String,Long>(**"sq"**,5L));  
  
 DataStream<Tuple2<String,Long>> datastream = env.fromCollection(source1);  
 DataStream<Tuple2<String,Long>> result = datastream.keyBy(0).countWindow(3)  
 .reduce(**new** ReduceFunction<Tuple2<String, Long>>() {  
 @Override  
 **public** Tuple2<String, Long> reduce(Tuple2<String, Long> t1, Tuple2<String, Long> t2) **throws** Exception {  
  
 **return new** Tuple2<String,Long>(t1.**f0**,t1.**f1**+t2.**f1**);  
 }  
 });  
 result.print();  
 env.execute(**"reduceFunction"**);  
 }  
}

### 3.AggregateFunction

AggregateFunction：更加通用，也更加复杂，通过WindowedStream的aggregate方法指定一个AggregateFunction来处理。其中实现AggregateFunction需要传入三个泛型，第一个表示源数据类型，第二个表示acc（accumulator）的类型，第三个是结果数据类型，并且要实现四个方法，createAccumulator为初始化acc，其目的是用于add第一个元素，add将每一个元素以某种方式添加到acc中，getResult获取最终计算结果，merge为合并acc；也就是说add需要传入一条元素和当前累加的中间结果，且第一次add的acc是预先定义的createAccumulator，add输出的是中间状态的acc，一般来说，元素add完毕之后便会调用getResult计算自身业务想要的结果。

**AggregateFunction是一个基于中间计算结果状态进行增量计算的函数，由于是迭代计算方式，所以，在窗口处理过程中，不用缓存整个窗口的数据，所以效率执行比较高。**

需要实现4个接口

(1)ACC createAccumulator(); 迭代状态的初始值

(2)ACC add(IN value, ACC accumulator); 每一条输入数据，和迭代数据如何迭代

(3)ACC merge(ACC a, ACC b); 多个分区的迭代数据如何合并

(4)OUT getResult(ACC accumulator); 返回数据，对最终的迭代数据如何处理，并返回结果。

开发：

**package** com.sqq2;  
  
**import** org.apache.flink.api.common.functions.AggregateFunction;  
**import** org.apache.flink.api.common.functions.ReduceFunction;  
**import** org.apache.flink.api.java.tuple.Tuple2;  
**import** org.apache.flink.streaming.api.datastream.DataStream;  
**import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;  
  
**import** java.util.ArrayList;  
**import** java.util.List;  
  
**public class** wr2 {  
 **public static void** main(String[] args) **throws** Exception {  
 StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  
 List source1 = **new** ArrayList();  
 source1.add(**new** Tuple2<String,Long>(**"sq"**,15L));  
 source1.add(**new** Tuple2<String,Long>(**"sq"**,10L));  
 source1.add(**new** Tuple2<String,Long>(**"sq"**,5L));  
  
 DataStream<Tuple2<String,Long>> datastream = env.fromCollection(source1);  
 DataStream<Double> result = datastream.keyBy(0).countWindow(3)  
 .aggregate(**new** AverageAggregate());  
 result.print();  
 env.execute(**"reduceFunction"**);  
 }  
 **public static class** AverageAggregate  
 **implements** AggregateFunction<Tuple2<String, Long>, Tuple2<Long, Long>, Double> {  
  
 @Override  
 **public** Tuple2<Long, Long> createAccumulator() {  
 **return new** Tuple2<>(0L,0L);  
 }  
  
 @Override  
 **public** Tuple2<Long, Long> add(Tuple2<String, Long> value, Tuple2<Long, Long> accumulator) {  
 **return new** Tuple2<>(accumulator.**f0** + value.**f1**, accumulator.**f1** + 1L);  
 }  
  
 @Override  
 **public** Double getResult(Tuple2<Long, Long> accumulator) {  
 **return** ((**double**) accumulator.**f0**) / accumulator.**f1**;  
 }  
  
 @Override  
 **public** Tuple2<Long, Long> merge(Tuple2<Long, Long> a, Tuple2<Long, Long> b) {  
 **return new** Tuple2<>(a.**f0** + b.**f0**, a.**f1** + b.**f1**);  
 }  
 }  
}

### 4.FlodFunction

FlodFunction：可以根据定义的规则将外部元素合并到窗口元素中。flink中已经Deprecated警告，且建议使用AggregateFunction代替。

开发：

**package** com.sqq2;  
  
**import** org.apache.flink.api.common.functions.FoldFunction;  
**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** java.util.ArrayList;  
**import** java.util.List;  
  
**public class** wr3 {  
 **public static void** main(String[] args) **throws** Exception {  
 StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  
 List source1 = **new** ArrayList();  
 source1.add(**new** Tuple2<String, Long>(**"sq"**, 15L));  
 source1.add(**new** Tuple2<String, Long>(**"sq"**, 10L));  
 source1.add(**new** Tuple2<String, Long>(**"sq"**, 5L));  
 DataStream<Tuple2<String, Long>> datasource1 = env.fromCollection(source1);  
 SingleOutputStreamOperator<String> fold = datasource1.keyBy(0).  
 countWindow(3).fold(**"haha"**, **new** FoldFunction<Tuple2<String, Long>, String>() {  
 @Override  
 **public** String fold(String accumulator, Tuple2<String, Long> value) **throws** Exception {  
 **return** accumulator+value.**f1**;  
 }  
 });  
 fold.print();  
 env.execute(**"fold"**);  
 }  
}

### 5.ProcessWindowFunction

在某些业务场景下，统计更复杂的指标，就可能会依赖窗口中所有的数据元素，以及可能会需要操作窗口中的状态数据和窗口元数据，全量聚合函数ProcessWindowFunction能够提供类似这种支持。ProcessWindowFunction的简单应用如：统计窗口数据元素中某一字段的中位数和众数。

（1）增量聚合: 窗口不维护原始数据，只维护中间结果，每次基于中间结果和增量数据进行聚合。如: ReduceFunction、AggregateFunction。

(2)全量聚合: 窗口需要维护全部原始数据，窗口触发进行全量聚合。如:ProcessWindowFunction。

#### 5.1 全量聚合ProcessWindowFunction

Flink针对全量聚合计算提供了一个骨架抽象类ProcessWindowFunction，如果我们不需要操作状态数据，则只需要实现ProcessWindowFunction的process（）方法即可，在该方法中具体定义计算评估和输出的逻辑。

开发：

**package** com.sqq2;  
  
  
**import** org.apache.flink.api.common.functions.MapFunction;  
**import** org.apache.flink.api.java.tuple.Tuple2;  
**import** org.apache.flink.api.java.tuple.Tuple5;  
**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.windowing.ProcessWindowFunction;  
  
**import** org.apache.flink.streaming.api.windowing.time.Time;  
**import** org.apache.flink.streaming.api.windowing.windows.TimeWindow;  
**import** org.apache.flink.util.Collector;  
  
  
**public class** wr4 {  
 **public static void** main(String[] args) **throws** Exception {  
  
 StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  
 DataStream<Tuple2<String, Long>> dataStreamSource = env.socketTextStream(**"192.168.8.201"**,9999).map(**new** MapFunction<String, Tuple2<String, Long>>() {  
 @Override  
 **public** Tuple2<String, Long> map(String s) **throws** Exception {  
 **return new** Tuple2<String,Long>(s.split(**" "**)[0],Long.*parseLong*(s.split(**" "**)[1]));  
 }  
 });  
 SingleOutputStreamOperator<Tuple5<String, Long, Long, Long, Long>> result = dataStreamSource.keyBy(t -> t.**f0**)  
 .timeWindow(Time.*seconds*(10))  
 .process(**new** ProcessWindowFunction1());  
 result.print();  
 env.execute(**"wr4"**);  
 }  
  
 **public static class** ProcessWindowFunction1 **extends** ProcessWindowFunction<Tuple2<String, Long>, Tuple5<String, Long, Long, Long, Long>, String, TimeWindow> {  
  
 @Override  
 **public void** process(String key, Context context, Iterable<Tuple2<String, Long>> elements, Collector<Tuple5<String, Long, Long, Long, Long>> out) **throws** Exception {  
 Long sum = 0L;  
 Long max = **null**;  
 Long min = **null**;  
 **for** (Tuple2<String, Long> element : elements) {  
 sum += element.**f1**;  
 **if** (max == **null**) {  
 max = element.**f1**;  
 }  
 **if** (min == **null**) {  
 min = element.**f1**;  
 }  
 **if** (max < element.**f1**) {  
 max = element.**f1**;  
 }  
 **if** (min > element.**f1**) {  
 min = element.**f1**;  
 }  
 }  
 *// 求取窗口结束时间* **long** winEndTime = context.window().getEnd();  
 *// 返回计算结果* out.collect(**new** Tuple5<>(key, sum, max, min, winEndTime));  
  
 }  
 }  
}

结果：

Nc窗口输入

sq 100

sq 200

sq 500

sq 50

Scala-client窗口查看

(sq,550,500,50,1584089920000)

#### 5.2增量聚合AggregateFunction 结合ProcessWindowFunction

开发：

**package** com.sqq2;  
  
**import** org.apache.flink.api.common.functions.AggregateFunction;  
**import** org.apache.flink.api.common.functions.MapFunction;  
**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.windowing.ProcessWindowFunction;  
**import** org.apache.flink.streaming.api.windowing.time.Time;  
**import** org.apache.flink.streaming.api.windowing.windows.TimeWindow;  
**import** org.apache.flink.util.Collector;  
  
**public class** wr5 {  
 **public static void** main(String[] args) **throws** Exception {  
 StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  
 DataStream<Tuple2<String, Long>> dataStreamSource = env.socketTextStream(**"192.168.8.201"**,9999).map(**new** MapFunction<String, Tuple2<String, Long>>() {  
 @Override  
 **public** Tuple2<String, Long> map(String s) **throws** Exception {  
 **return new** Tuple2<String,Long>(s.split(**" "**)[0],Long.*parseLong*(s.split(**" "**)[1]));  
 }  
 });  
 SingleOutputStreamOperator<Tuple2<String, Double>> result = dataStreamSource.keyBy(t -> t.**f0**).  
 timeWindow(Time.*seconds*(10)).aggregate(**new** Avg1(), **new** pw1());  
 result.print();  
 env.execute(**"wr5"**);  
 }  
 **public static class** Avg1 **implements** AggregateFunction<Tuple2<String, Long>, Tuple2<Long, Long>, Double>{  
 @Override  
 **public** Tuple2<Long, Long> createAccumulator() {  
 **return new** Tuple2<>(0L, 0L);  
 }  
  
 @Override  
 **public** Tuple2<Long, Long> add(Tuple2<String, Long> value, Tuple2<Long, Long> accumulator) {  
 **return new** Tuple2<Long, Long>(accumulator.**f0** + value.**f1**, +accumulator.**f1** + 1);  
 }  
  
 @Override  
 **public** Double getResult(Tuple2<Long, Long> accumulator) {  
 **return** ((**double**) accumulator.**f0**) / accumulator.**f1**;  
 }  
  
 @Override  
 **public** Tuple2<Long, Long> merge(Tuple2<Long, Long> a, Tuple2<Long, Long> b) {  
 **return new** Tuple2<Long, Long>(a.**f0** + b.**f0**, +a.**f1** + b.**f1**);  
 }  
 }  
 **private static class** pw1 **extends** ProcessWindowFunction<Double, Tuple2<String, Double>, String, TimeWindow> {  
 **public void** process(String key,  
 Context context,  
 Iterable<Double> averages,  
 Collector<Tuple2<String, Double>> out) {  
 Double average = averages.iterator().next();  
 out.collect(**new** Tuple2<>(key, average));  
 }  
 }  
}