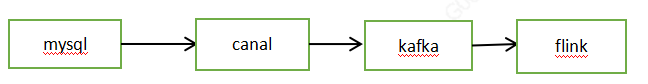
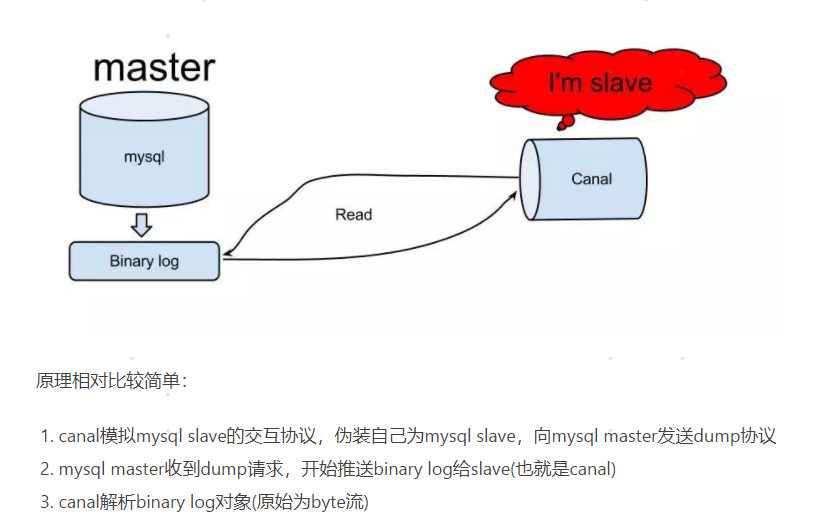
**实时计算开发流程**

基于masql->canal->kafka->Flink的数据实时增量同步



**canal**



**canal安装部署**

下载地址

<https://github.com/alibaba/canal/releases/download/canal-1.1.4/canal.adapter-1.1.4.tar.gz>

参考文档

<https://github.com/alibaba/canal/wiki/Canal-Kafka-RocketMQ-QuickStart>

修改conf/canal.properties

#zk地址

canal.zkServers = 127.0.0.1:2181

# tcp, kafka, RocketMQ

canal.serverMode = kafka

#mq地址

canal.mq.servers = 127.0.0.1:9092

canal.mq.producerGroup = test

修改conf/example/instance.properties

## mysql serverId

canal.instance.mysql.slaveId = 1234

#数据库地址

canal.instance.master.address = 127.0.0.1:3306

#数据库账号密码

canal.instance.dbUsername = canal

canal.instance.dbPassword = canal

# 只监控iwater\_screen库下的所有表

canal.instance.filter.regex=iwater\_screen\..\*

# kafka动态topic=库名.表名

canal.mq.dynamicTopic=iwater\_screen\\..\*

启动

cd /usr/local/canal/ sh

bin/startup.sh

查看日志

a.查看 logs/canal/canal.log

vi logs/canal/canal.log

b. 查看instance的日志：

vi logs/example/example.log

关闭

cd /usr/local/canal/

sh bin/stop.sh

**mysql配置**

* 对于自建 MySQL , 需要先开启 Binlog 写入功能，配置 binlog-format 为 ROW 模式，my.cnf 中配置如下

[mysqld]

log-bin=mysql-bin # 开启 binlog

binlog-format=ROW # 选择 ROW 模式

server\_id=1 # 配置 MySQL replaction 需要定义，不要和 canal 的 slaveId 重复

* + 注意：针对阿里云 RDS for MySQL , 默认打开了 binlog , 并且账号默认具有 binlog dump 权限 , 不需要任何权限或者 binlog 设置,可以直接跳过这一步
* 授权 canal 链接 MySQL 账号具有作为 MySQL slave 的权限, 如果已有账户可直接 grant

CREATE USER canal IDENTIFIED BY 'canal';

GRANT SELECT, REPLICATION SLAVE, REPLICATION CLIENT ON \*.\* TO 'canal'@'%';

-- GRANT ALL PRIVILEGES ON \*.\* TO 'canal'@'%' ;

FLUSH PRIVILEGES;

**kafka**

下载

<https://www.apache.org/dyn/closer.cgi?path=/kafka/1.0.0/kafka_2.11-1.0.0.tgz>

单机部署

修改 /config/server.properties

#ip为当前机器主机，9092为kafka端口号

advertised.listeners=PLAINTEXT://118.190.174.193:9092

#修改日志路径

log.dirs=/opt/kafka\_2.11-1.0.0/logs

#修改zk地址

zookeeper.connect=localhost:2181

启动kafka

./bin/kafka-server-start.sh ./config/server.properties &

创建topie

./bin/kafka-topics.sh --create --zookeeper

查看指定topic消费数据， test 为要查看的topic

./bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic test --from-beginning

发送消息

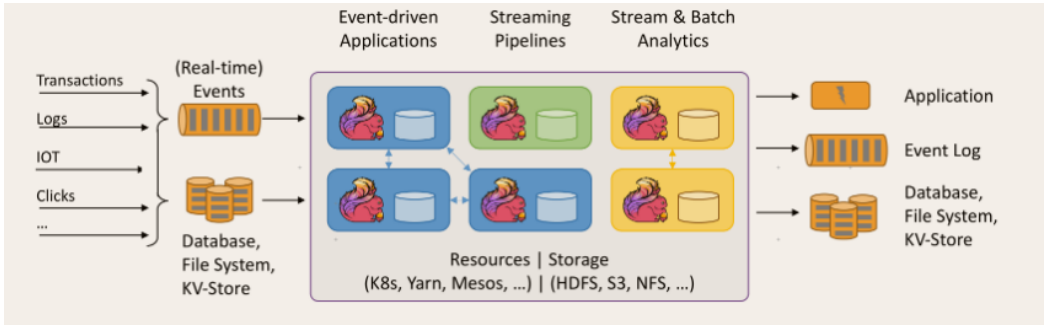
./bin/kafka-console-producer.sh --broker-list localhost:9092 --topic test

**flink**



Apache Flink 是为分布式、高性能、随时可用以及准确 的流处理应用程序打造的开源流处理框架。

Apache Flink 是一个框架和分布式处理引擎，用于对无界和有界数据流进行有状态计算。Flink 被设计在所有常见的集群环境中运行，以内存执行速度和任意规模来执行计算。



**流处理与批处理**

批处理的特点是有界、持久、大量，非常适合需要访问全套记录才能完成的计 算工作，一般用于离线统计。

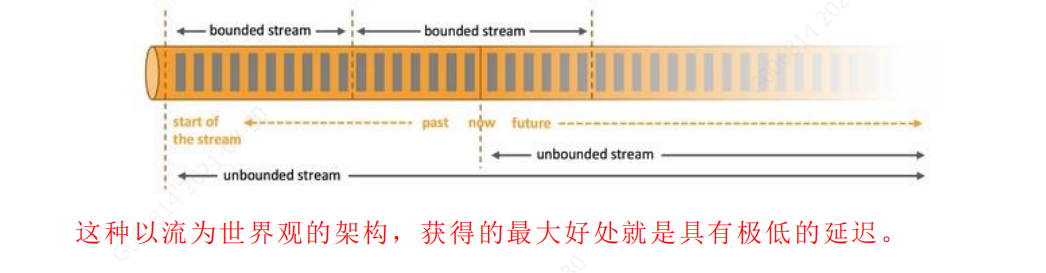
流处理的特点是无界、实时, 无需针对整个数据集执行操作，而是对通过系统 传输的每个数据项执行操作，一般用于实时统计。

在 spark 的世界观中，一切都是由批次组成的，离线数据是一个大批次，而实时数据是由一个一个无限的小批次组成的。

而在 flink 的世界观中，一切都是由流组成的，离线数据是有界限的流，实时数据是一个没有界限的流，这就是所谓的有界流和无界流。

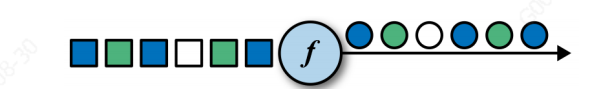
**无界数据流：无界数据流有一个开始但是没有结束，它们不会在生成时终止并提供数据，必须连续处理无界流，也就是说必须在获取后立即处理 event。对于无界数据流我们无法等待所有数据都到达，因为输入是无界的，并且在任何时间点都不会完成。处理无界数据通常要求以特定顺序（例如事件发生的顺序）获取 event，以 便能够推断结果完整性。**

**有界数据流：有界数据流有明确定义的开始和结束，可以在执行任何计算之前通过获取所有数据来处理有界流处理有界流不需要有序获取，因为可以始终对有界数据集进行排序，有界流的处理也称为批处理。**



**算子 DataStream stream**

map(数据类型转换) 数据处理后返回的是单条记录



DataStream<Integer> mapStram = dataStream.map(new MapFunction<String, Integer>() {

public Integer map(String value) throws Exception {

return value.length();

}

});

flatMap(数据类型转换) 数据处理后返回的是多条数据，要用collect处理

DataStream<String> flatMapStream = dataStream.flatMap(new FlatMapFunction<String,

String>() {

public void flatMap(String value, Collector<String> out) throws Exception {

String[] fields = value.split(",");

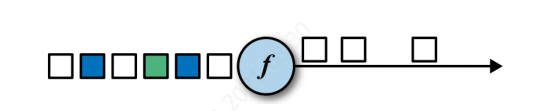
for( String field: fields )

out.collect(field);

}

});

filter(数据筛选)，筛选sensor\_1开头的id对应的数据



DataStream<String> mapStream=stream.filter(new FilterFunction<String>() {

@Override

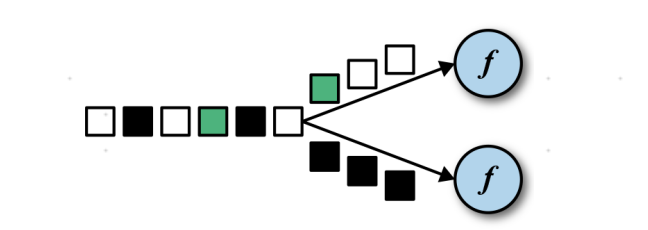
public boolean filter(String s) throws Exception {

return s.startsWith("sensor\_1");

}

});

keyBy 将一个流拆分成不相交的分区，每个分区包含具有相同的key的元素，在内部以hash的形式实现的



sum()

min() 返回最小值

max() 返回最大值

minBy() 返回最小值所在的整条数据

maxBy() 返回最大值所在的整条数据

DataStream<FlumeTest> dataStream=stream.map(line->{

JSONObject object = JSONObject.parseObject(line);

return new FlumeTest(JSONObject.parseObject(object.toString()).getInteger("id"),

JSONObject.parseObject(object.toString()).getString("ownership"),

JSONObject.parseObject(object.toString()).getDoubleValue("state"));

});

//分组

KeyedStream<FlumeTest, Tuple> keyedStream=dataStream.keyBy("ownership");

KeyedStream<FlumeTest, String> keyedStream1=dataStream.keyBy(data->data.getOwnership());

//滚动聚合

//求最大值，state里返回的一直是最大值，其他字段返回当前这条数据的值

SingleOutputStreamOperator<FlumeTest> resultStream=keyedStream.max("state");

//求最大值，所有字段返回的都是最大值的那条数据

SingleOutputStreamOperator<FlumeTest> resultStream1=keyedStream.maxBy("state");

Reduce

KeyedStream->DataStream:一个分组数据流的聚合操作，合并当前的元素和上次聚合的结果，产生一个新的值，返回的流中包含每一次聚合的结果，而不是只返回最后一次聚合的最终结果。

//取最大的state值，以及当前最新的时间戳

keyedStream.reduce(new ReduceFunction<FlumeTest>(){

@Override

//t1 是当前最新的数据，t2之前聚合的结果状态，返回的是最新的状态

public FlumeTest reduce(FlumeTest t2, FlumeTest t1) throws Exception {

//取最大的state值

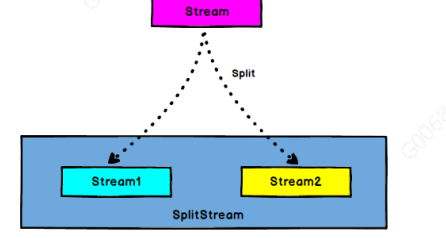
return new FlumeTest(t1.getId(),t1.getOwnership(),Math.max(t2.getState(),t1.getState()),t1.getTimesTamp());

}

});

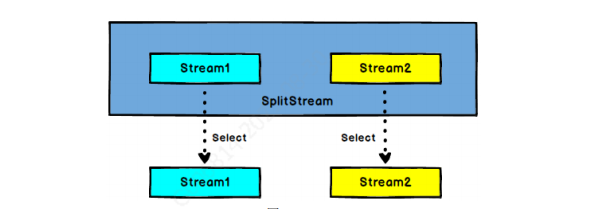
split

DataStream->SplitStream 根据某些特征把一个DataStream拆分成两个或者多个DataStream



select

SplitStream->DataStream 从一个splitStream中获取一个或者多个DataStream



//分流，按照state值10为分界分为两条流

SplitStream<FlumeTest> splitStream= dataStream.split(new OutputSelector<FlumeTest>(){

@Override

public Iterable<String> select(FlumeTest flumeTest) {

return (flumeTest.getState()>10)? Collections.singletonList("high"):Collections.singletonList("low");

}

});

DataStream<FlumeTest> highTempStream=splitStream.select("high");

DataStream<FlumeTest> lowTempStream=splitStream.select("low");

DataStream<FlumeTest> allTempStream=splitStream.select("high","low");

connect DataStream,DataStream->ConnectedStream 连接两个保持他们类型的数据流， 两个数据流被Connect之后，只是被放在了同一个流中，内部依然保持各自的数据和形式不发生任何变化，两个流相互独立。

CoMap,CoFlatMap 数据类型转换相当与map和flatMap

//合流connect，将highTempStream转换成二元组类型，与lowTempStream连接合并之后，输出状态信息

DataStream<Tuple2<String,Double>> warningStream=highTempStream.map(new MapFunction<FlumeTest, Tuple2<String, Double>>() {

@Override

public Tuple2<String, Double> map(FlumeTest flumeTest) throws Exception {

return new Tuple2<>(flumeTest.getOwnership(),flumeTest.getState());

}

});

ConnectedStreams<Tuple2<String,Double>,FlumeTest> connectedStreams=warningStream.connect(lowTempStream);

DataStream<Object> coMapStream=connectedStreams.map(new CoMapFunction<Tuple2<String, Double>, FlumeTest, Object>() {

@Override

public Object map1(Tuple2<String, Double> stringDoubleTuple2) throws Exception {

return new Tuple3<>(stringDoubleTuple2.f0,stringDoubleTuple2.f1,"high");

}

@Override

public Object map2(FlumeTest flumeTest) throws Exception {

return new Tuple2<>(flumeTest.getOwnership(),flumeTest.getState());

}

});

union 对两个或者两个以上的DataStream进行union操作，产生一个包含所有DataStream元素的新DataStream

//union联合多条流，只能联合数据类型相同的流

DataStream<FlumeTest> unionStream=highTempStream.union(lowTempStream,allTempStream);

**Window 窗口函数**

**Window 概述**

streaming 流式计算是一种被设计用于处理无限数据集的数据处理引擎，而无限数据集是指一种不断增长的本质上无限的数据集，而 window 是一种**切割无限数据 为有限块进行处理的手段**。

Window 是无限数据流处理的核心，Window 将一个无限的 stream 拆分成有限大小的”buckets”桶，我们可以在这些桶上做计算操作。

**Window 类型**

Window 可以分成两类：

➢ CountWindow：按照指定的数据条数生成一个 Window，与时间无关。

➢ TimeWindow：按照时间生成 Window。

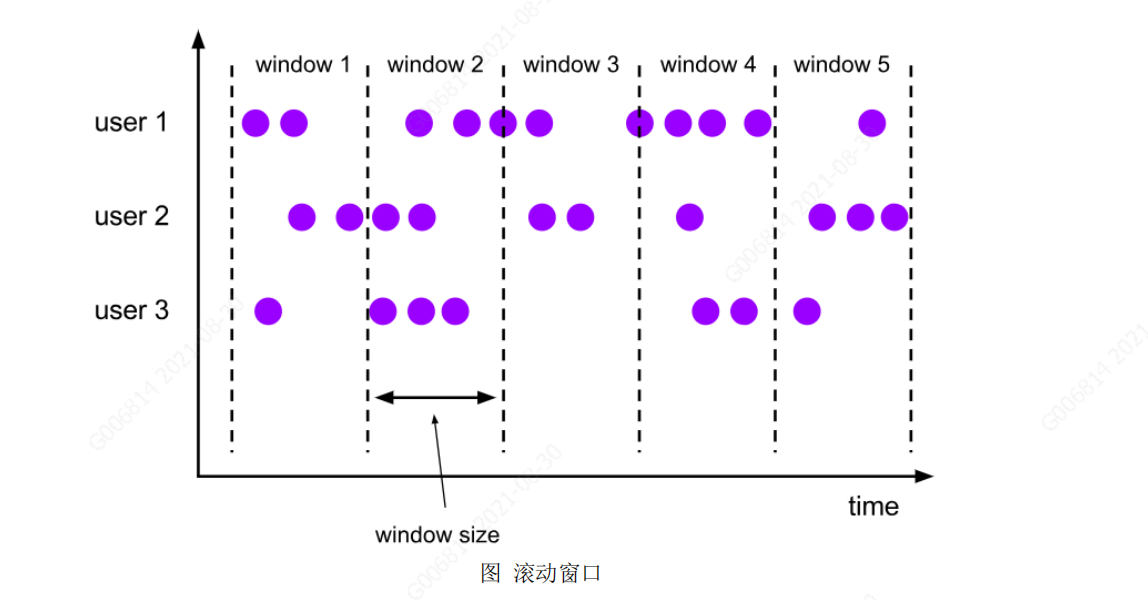
对于 TimeWindow，可以根据窗口实现原理的不同分成三类：滚动窗口（Tumbling Window）、滑动窗口（Sliding Window）和会话窗口（Session Window）。

**1. 滚动窗口（Tumbling Windows）**

将数据依据固定的窗口长度对数据进行切片。

特点：时间对齐，窗口长度固定，没有重叠。

滚动窗口分配器将每个元素分配到一个指定窗口大小的窗口中，滚动窗口有一 个固定的大小，并且不会出现重叠。例如：如果你指定了一个 5 分钟大小的滚动窗 口，窗口的创建如下图所示：



适用场景：适合做 BI 统计等（做每个时间段的聚合计算）。

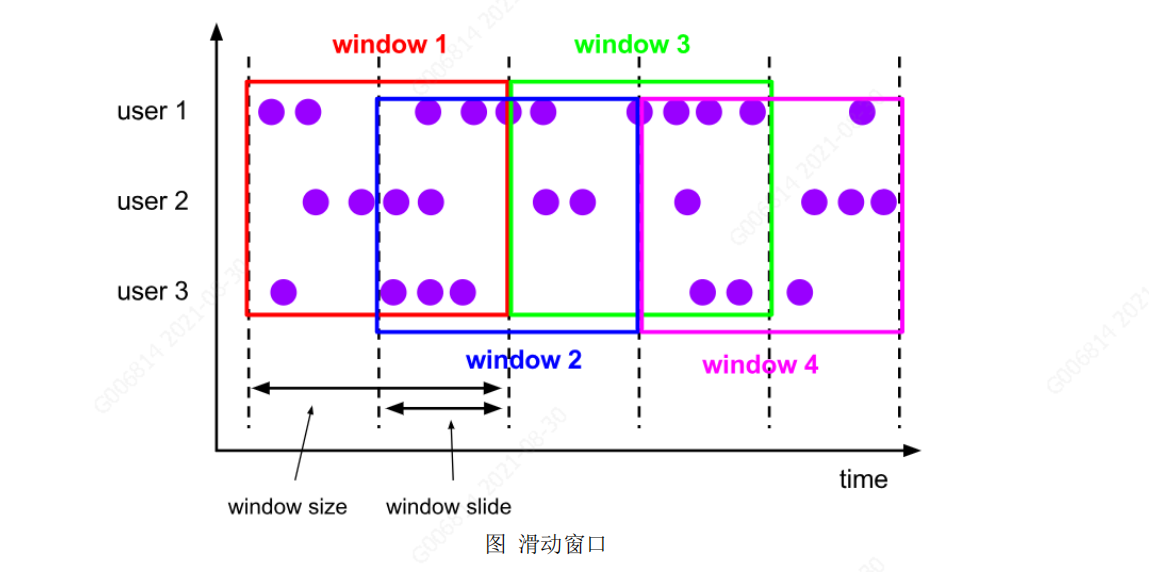
**2. 滑动窗口（Sliding Windows）**

滑动窗口是固定窗口的更广义的一种形式，滑动窗口由固定的窗口长度和滑动间隔组成。

特点：时间对齐，窗口长度固定，可以有重叠。

滑动窗口分配器将元素分配到固定长度的窗口中，与滚动窗口类似，窗口的大 小由窗口大小参数来配置，另一个窗口滑动参数控制滑动窗口开始的频率。因此， 滑动窗口如果滑动参数小于窗口大小的话，窗口是可以重叠的，在这种情况下元素 会被分配到多个窗口中。

例如，你有 10 分钟的窗口和 5 分钟的滑动，那么每个窗口中 5 分钟的窗口里包 含着上个 10 分钟产生的数据，如下图所示：



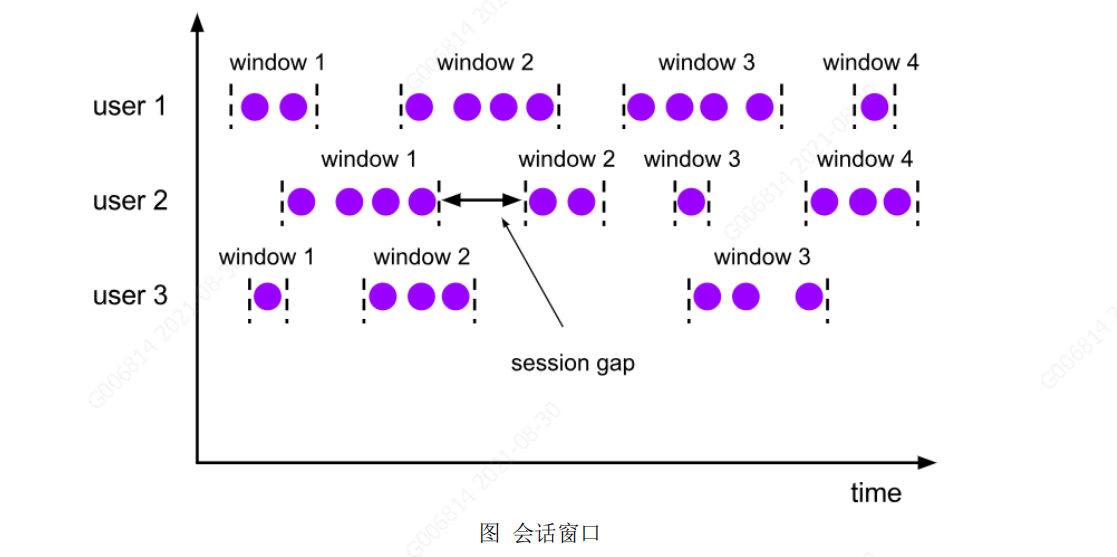
适用场景：对最近一个时间段内的统计（求某接口最近 5min 的失败率来决定是否要报警）。

**3. 会话窗口（Session Windows）**

由一系列事件组合一个指定时间长度的 timeout 间隙组成，类似于 web 应用的session，也就是一段时间没有接收到新数据就会生成新的窗口。

特点：时间无对齐。

session 窗口分配器通过 session 活动来对元素进行分组，session 窗口跟滚动窗口和滑动窗口相比，不会有重叠和固定的开始时间和结束时间的情况，相反，当它在一个固定的时间周期内不再收到元素，即非活动间隔产生，那个这个窗口就会关闭。一个 session 窗口通过一个 session 间隔来配置，这个 session 间隔定义了非活跃 周期的长度，当这个非活跃周期产生，那么当前的 session 将关闭并且后续的元素将被分配到新的 session 窗口中去。



**Window API**

**TimeWindow**

TimeWindow 是将指定时间范围内的所有数据组成一个 window，一次对一个window 里面的所有数据进行计算。

**1. 滚动窗口**

Flink 默认的时间窗口根据 Processing Time 进行窗口的划分，将 Flink 获取到的数据根据进入 Flink 的时间划分到不同的窗口中。

DataStream<Tuple2<String, Double>> minTempPerWindowStream = dataStream

.map(new MapFunction<SensorReading, Tuple2<String, Double>>() {

@Override

public Tuple2<String, Double> map(SensorReading value) throws Exception {

return new Tuple2<>(value.getId(), value.getTemperature());

}

})

.keyBy(data -> data.f0)

.timeWindow( Time.seconds(15) )

.minBy(1);

时间间隔可以通过 Time.milliseconds(x)，Time.seconds(x)，Time.minutes(x)等其中的一个来指定。

**2. 滑动窗口（SlidingEventTimeWindows）**

滑动窗口和滚动窗口的函数名是完全一致的，只是在传参数时需要传入两个参数，一个是 window\_size，一个是 sliding\_size。

下面代码中的 sliding\_size 设置为了 5s，也就是说，每 5s 就计算输出结果一次，每一次计算的 window 范围是 15s 内的所有元素。

DataStream<SensorReading> minTempPerWindowStream = dataStream

.keyBy(SensorReading::getId)

.timeWindow( Time.*seconds*(15), Time.*seconds*(5) )

.minBy(**"temperature"**);

时间间隔可以通过 Time.milliseconds(x)，Time.seconds(x)，Time.minutes(x)等其中的一个来指定。

**CountWindow**

CountWindow 根据窗口中相同 key 元素的数量来触发执行，执行时只计算元素数量达到窗口大小的 key 对应的结果。

注意：CountWindow 的 window\_size 指的是相同 Key 的元素的个数，不是输入

的所有元素的总数。

1. **滚动窗口**

默认的 CountWindow 是一个滚动窗口，只需要指定窗口大小即可，当元素数量达到窗口大小时，就会触发窗口的执行。

DataStream<SensorReading> minTempPerWindowStream = dataStream

.keyBy(SensorReading::getId)

.countWindow( 5 )

.minBy(**"temperature"**);

**2. 滑动窗口**

滑动窗口和滚动窗口的函数名是完全一致的，只是在传参数时需要传入两个参数，一个是 window\_size，一个是 sliding\_size。下面代码中的 sliding\_size 设置为了 2，也就是说，每收到两个相同 key 的数据就计算一次，每一次计算的 window 范围是 10 个元素。

DataStream<SensorReading> minTempPerWindowStream = dataStream

.keyBy(SensorReading::getId)

.countWindow( 10, 2 )

.minBy(**"temperature"**);

​